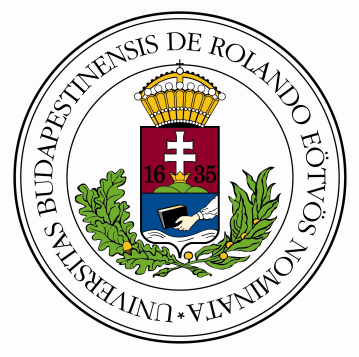
Tools and Language Elements for Testing, Encapsulation and Controlling Abstraction in Large Scale C++ Projects

Doctoral dissertation
2019

Gábor Márton
martong@caesar.elte.hu



Thesis advisor: **Dr Zoltán Porkoláb, docent**
Eötvös Loránd University, Faculty of Informatics,
1117 Budapest, Pázmány Péter sétány 1/C

ELTE IK Doctoral School
Doctoral program: Foundations and Methodologies of Informatics
Head of the doctoral school: Prof. Dr Erzsébet Csuhaj-Varjú
Head of the doctoral program: Prof. Dr Zoltán Horváth

Contents

List of Figures

Acknowledgements

I would like to thank wholeheartedly to my wife Barbara, who supported me with her patience and empathy throughout all the years I have spent working on this thesis. This work demanded plenty of my free-time, evenings and weekends which I could have spent rather with her. My heartfelt thanks go to my parents for all their care and help. I'd also like to thank my brother, his family and his kids. They always helped me to get some refreshing fun, so I could break the monotony of the research.

Of course, thanks to all the people who advised me with their technical expertise. My special thanks go to Zoltán Porkoláb for his counsel, ideas and all those hours he has spent with the reviews. In addition, thanks to my colleges, Imre Szekeres, Máté Cserna, Péter Bolla, Gábor Horváth, Richárd Szalay, Péter Szabados, Mátyás Végh, Zoltán Gera, Máté Csákvári and Krisztián Pándi for the valuable discussions we had on the different research topics.

Chapter 1

Introduction

Professional software development involves *testing* as a quintessential constituent of the development process [22]. From the basic unit tests through the integration tests up to the high-level end-to-end tests (which sit at the top of the test pyramid [23]) their importance in quality assurance and in continuous integration/deployment is indisputable [24, 25].

One important testing strategy is *black-box* (also known as specification-based, or input/output-driven) testing. During this testing method, we view the program itself as a black box. That is, we are completely unconcerned about the internal behaviour and structure of the program. Instead, we concentrate on finding circumstances in which the program does not behave accordingly to its specifications [22]. Another testing strategy is *white-box* testing (also known as structure-based testing), which permits us to examine the internal structure of the program. This strategy derives test data from an inspection of the program's logic. One goal of these kinds of tests is to exercise paths of control flow through the code. If we execute all possible paths of control flow through the program then possibly the program has been completely tested, and this is considered to be exhaustive path testing, which is impossible in practice.

Test-driven development (TDD) is a software development practice which requires writing automated tests prior to developing any functional code. The development consists of very short iterations of writing a new test and then providing an implementation for it [26, 27]. TDD involves that first we create a contract (an interface) between the component and the tests. In one iteration, we write a test which uses the interface and describes one behaviour. At this point, the test may fail, since the implementation does not provide the desired behaviour. As the next phase of the iteration, we provide the implementation in a way that both the preexisting tests and the newly added test pass. This method implicates that all tests are black-box tests. We may find ourselves in a situation where we have to substitute (or mock) something that the implementation depends on and what

is never exposed via the public interface. According to the TDD principles, this means that we have made a mistake and the contract should be widened to expose that internal dependency via the interface. However, *encapsulation* is another very strong principle in software development which may be violated if we give away such internals [28]. By widening the public interface and giving away strongly coupled internals, the previous code structure deteriorates. This perception is aligned with the empirical results of M. Siniaalto and P. Abrahamsson, who reveal that unwanted side effect of TDD can be that some parts of the code may deteriorate [29]. Consequently, some tests may not apply to this interface widening principle because the priority of encapsulation may be higher for the given component. Even if we do not expose the internal dependencies of a component, there may be some dependencies which should be replaced in order to create meaningful tests. There are existing techniques to substitute these dependencies without intrusively changing (widening) the public interface (e.g. in C/C++, one such method is when we use the linker to do the replacement). However, all these existing techniques have definite drawbacks.

Often, legacy code bases evolved without having very few (or not at all any) automated tests. According to Michael Feathers, the main thing that distinguishes legacy code from non-legacy code is a lack of comprehensive tests [30]. Refactoring such code in order to provide tests is very hard because we cannot verify correctness without having tests; hence it is a vicious circle. Nonetheless, we can break the circle with such tests which does not modify the existing production code. In this dissertation, we refer to tests which do not require any modification in the production code as *non-intrusive* tests. Non-intrusive tests may be used both in the case when encapsulation has a priority over TDD and when we would like to add tests for legacy code bases. There are existing techniques to support such non-intrusive tests (e.g. we may use the linker), but they all have some disadvantages. This dissertation provides new alternative techniques to write non-intrusive tests with significant advantages compared to earlier efforts (Thesis 1)

Encapsulation is one of the fundamental concepts in object-oriented programming [31], we referred to it before. It is used to distinguish between the interface and the implementation of a class, thus minimizing the dependencies among separately-written modules [32, 33]. The interface is regularly defined in terms of possible services or methods the class offers to its clients. Methods are specified in forms of their signature. The implementation of a class defines the internal representation of its objects and the way its methods are implemented. The main idea behind encapsulation is to hide as many details of the representation of the classes as possible. This way we can greatly reduce coupling between objects of different classes, enforce a precise definition of class interfaces, and increase reusability. However, sometimes we have to break encapsulation if we would like to create

white-box tests, because white-box testing may require the ability to access private data and implementation details. In some compiled languages like C/C++, it is not trivial to access these private internals, as we will see in Chapter 2.5. In this dissertation, we provide new techniques to access private members and methods in the test while preserving encapsulation in the production code (Thesis 2).

Encapsulation may be weakened also if one class has uncontrolled access to all internal members of another class. The `friend` language construct in C++ provides such access. Thus, C++ friends are often criticised. On the other hand, some researchers claim that if friends are used judiciously then they may be a better choice than widening the public interface of a class [34]. We investigated the use of friends in several open source projects and this dissertation presents our results. These results have motivated us to propose a *selective friend* language construct for C++ which can restrict friendship only to well-defined members (Thesis 3). Such a new language element may decrease the degradation of encapsulation and significantly increase the diagnostic capacity of the compiler.

Besides encapsulation, another quintessential concept of software development is *abstraction*. Architectures for software use abstractions (idioms) to describe system components, the nature of interactions among the components, and the patterns that guide the composition of components into systems [35]. Raising the level of abstraction reduces the complexity and brittleness of software through encapsulation. Powerful abstractions that encapsulate large amounts of low-level code tend to address highly specialized domains [36]. One such domain is *concurrency*. For instance, in C++, a `lock_guard` is a high-level abstraction, a mutex wrapper which provides a mechanism for owning a mutex for the duration of a scoped block [37]. Using the `lock_guard` instead of low-level mutex functions reduces complexity and makes the software less prone to errors. Actually, the `lock_guard` is one example of the "Resource Acquisition Is Initialization" (RAII) idiom [38]. *Read-copy-update* (RCU) is a concurrent programming technique to exchange data between threads [39]. It provides very efficient communication between threads if reads are way more frequent than writes. RCU is a frequently used concurrent pattern in low level, performance critical applications, like the Linux kernel (in fact, it originates from the Linux kernel). Earlier, there was no widely accepted high-level abstraction of RCU for the C++ programming language. In this dissertation, we present a new C++ class library to support the read-copy-update pattern (Thesis 4). The library has been carefully designed to optimise performance in a heavily multithreaded environment, at the same time providing high-level abstraction, like the `lock_guard` or smart pointers.

1.1 Thesis Structure

The next Chapter (2) describes and elaborates the foundations of non-intrusive testing in C/C++. We overview existing methods and their advantages and disadvantages. Two out of four theses are related to non-intrusive testing. Section 2.3 (as the first thesis) introduces new techniques for dependency replacement and non-intrusive testing. Non-intrusive tests often require access to private internals of a component, thus the different access methods are analysed in Section 2.5 and several new methods are introduced (the second thesis). Chapter 3 presents research results about how friends are used on open source projects and it describes the design and implementation of C++ selective friends (third thesis). Our higher-level abstraction for the read-copy-update pattern is introduced and analysed in Chapter 4 (fourth thesis). Lastly, Chapter 5 summarizes the contributions.

Chapter	Content
1	Define context and focus
2	Foundations for non-intrusive testing, Theses 1 and 2
2.3	Thesis 1: New non-intrusive testing methods
2.5	Thesis 2: Extending access for non-intrusive and white-box testing
3	Thesis 3: Selective friend
4	Thesis 4: High-level abstraction for the read-copy-update pattern
5	Summary

Table 1.1: Chapters and their content.

Chapter 2

Non-intrusive Testing

In C/C++, test code often influences the source code of the unit we want to test. During the test development process, we often have to introduce new interfaces to replace existing dependencies, e.g. a supplementary setter or constructor function has to be added to a class for dependency injection. In many cases, extra template parameters are used for the same purpose. These solutions may have serious detrimental effects on the code structure and sometimes on the run-time performance as well. We can use *non-intrusive* tests (tests which do not require any modification in the production code) to avoid these disadvantages. In this dissertation, we refer to all those testing approaches which require source code modification as *intrusive* testing.

Also, in legacy code bases often there are few or no unit tests. Refactoring such code in order to provide tests is almost impossible because we cannot verify correctness without having unit tests; hence it is a vicious circle. We can break the circle with non-intrusive tests, i.e. without actually modifying the production code.

Different non-intrusive testing methods have different weaknesses. In this chapter, we introduce new non-intrusive testing approaches which have clear advantages and complement the existing techniques.

2.1 Motivation

Testing is essential in modern software development [40, 41, 42, 43] to improve the quality of a system and reduce the cost of maintenance. There are different levels of testing from unit tests via integration tests to functional and non-functional tests. In this chapter, we focus on unit testing, which is the most language-specific method. However, some of the findings we discuss might be extended to different/higher level tests as well.

During a unit test, we check the behaviour of the unit under test. If we do functional programming and work with pure functions alone (where all functions are free from side-effects) then testing is easy because we just provide specific input and assert for the desired output. However, in the object-oriented paradigm, we have objects with some kind of state. This means the object is usually dependent on other objects that represent the internal state. Testing our object using these dependencies may be problematic; e.g. the dependency may represent a database or a network connection, whose behaviour can be hard or expensive to simulate. In order to create independent, resilient and efficient tests [44] in most cases we need to substitute some (or even all) of the dependencies with test doubles. In this dissertation, we call this substitution of dependencies as *dependency replacement*.

In object-oriented programming languages, the dependency replacement often requires the modification of the original public interface of the unit under test. For instance, new setter or constructor functions have to be added to a class, otherwise, dependency replacement would not work. Nevertheless, there are cases where these new functions are not intended to be used in production code. Moreover, in C++, source code modification for testing could result in performance degradation, e.g. introducing a new runtime interface and virtual functions just because of testing might worsen the performance of the production code. Also, in legacy code bases often there are no unit tests. Refactoring such legacy code in order to provide tests is almost impossible because we cannot verify correctness without having unit tests; hence it is a vicious circle. We can break the circle with non-intrusive tests, though all of the existing non-intrusive testing methods for C++ have some drawbacks.

2.2 Dependency Replacement in C++

Figure 2.1 shows a typical object under test, its dependencies and their possible replacements. If A and B are objects and "A depends on B", then we say that A is a *dependant* of B and B is a *dependency* of A. As for dependency replacement the dependant object is referred as the *system under test* (SUT). Sometimes we refer to that as the *unit* under test. In this study, we use the following definitions for test doubles: *Fake* classes provide empty definitions of functions in a way that the unit tests can pass. Fakes are the simplest doubles to cut down dependencies. *Stub* classes implement additionally some very basic behaviour, therefore they may be more complex than fakes. We can set up a stub to return with a specific value. *Mock* classes are used to formulate expectations, such as how many times a member function is called with a certain value.

There are several design patterns for dependency replacement like the Factory Method and Abstract Factory [45], the Service Locator [46, 47] and the Depen-

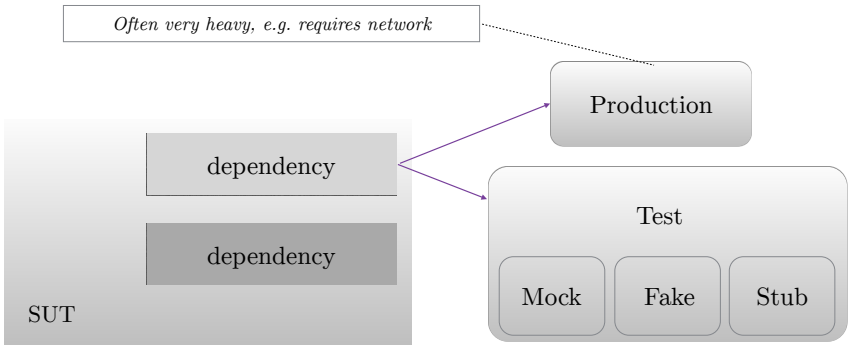


Figure 2.1: Dependency Replacement

dency Injection (DI) [48, 49, 50].

It is important to emphasize that dependency injection is different from the abstract concept of dependency replacement. Dependency injection (DI) is one realization – amongst many others – of dependency replacement. DI is used mostly in object-oriented languages with runtime reflection, like Java and C#. Most of these patterns can be used in the popular, managed languages and in C++ as well. Java and C# provide well documented DI frameworks (like the *Unity Container* in C# [51], and the Spring framework in Java [52]), therefore DI is the widespread method for performing dependency replacement in these managed languages. However, at the moment there is no generally accepted DI framework for C++.

Objects in the context of OOP are represented by instances of classes in C++. Since C++ is not a strict object-oriented language, we should also investigate other language constructs like free functions and function templates from the viewpoint of dependency replacement. Generally speaking, a dependant C++ entity (class, function, class template or function template) may have different (sometimes multiple) kinds of dependencies. For instance, it may depend on

- a global object (e.g. via a singleton).
- a global function (via a function call),
- an object via a pointer or reference,
- a type (e.g. via a type template parameter, or the type of a member)

2.2.1 C++ Seams

A *seam* is an abstract concept introduced by Feathers [30] as an instrument via we can alter behaviour without changing the original unit. Dependency replacement

is done via seams in C++. Up to now, four seams have been identified in the literature [53, 54]:

1. *Link seam*: Change the definition of a function via some linker specific setup.
2. *Preprocessor seam*: With the help of the preprocessor, redefine function names to use an alternative implementation.
3. *Object seam*: Based on inheritance to inject a subclass with an alternative implementation.
4. *Compile seam*: Inject dependencies at compile-time through template parameters.

The *enabling point* of a seam is the place where we can make the decision to use one behaviour or another. Different seams have different enabling points. For example, replacing the constructor argument for the implementation of an interface with a mock implementation when a unit test is set up is an object seam with the constructor as an enabling point.

Link Seam

We can use a link seam e.g. to replace the implementation of a free function or a member function as presented in Figure 2.2. we should link the production

```
// A.hpp
void foo();
// A.cpp
void foo() { ... };
// MockA.cpp
void foo() { ... };
// B.cpp
#include "A.hpp"
void bar() { foo(); ... }
```

Figure 2.2: An example of a link seam

code with `A.o`. On the other hand, when we test the `bar()` function then we should link the test executable to the `MockA.o` object file. Link-time dependency replacement is not possible if the dependency is defined in a static library or in the same translation unit where the SUT is defined. It is also not feasible to use link seams if the dependency is implemented as an inline function [53]. This makes the use of this seam cumbersome or practically impossible when the dependant unit is a template or when the dependency is a template. The enabling point for a link seam is always outside of the program text (in many cases hidden in the build system). Link seams keep the test setup code separated from the other phases of the test. This makes the use of link seams quite difficult to identify. On top

of all that, link-time substitution requires strong support from the build system we are using. Thus, we might have to specialize the building of the tests for each and every unit. This does not scale well and can be really demanding regarding maintenance.

Preprocessor Seam

Preprocessor seams can be applied to replace the invocation of a global function to an invocation of a test double [55]. Let us consider the code snippet in Figure 2.3. We can replace the standard `malloc()` and `free()` functions with our own

```
void *my_malloc(size_t size) {
  //...
  return malloc(size);
}

void my_free(void *p) {
  //...
  return free(p);
}

#define free my_free
#define malloc my_malloc

void unitUnderTest() {
  int *array = (int *)malloc(4 * sizeof(int));
  // do something with array
  free(array);
}
```

Figure 2.3: An example of a preprocessor seam

implementation. One example usage is to collect statistics or do sanity checks with `my_malloc` and `my_free` functions. These seams can be applied conveniently in C, but not in C++. As soon as we use namespaces, the preprocessor might generate code which cannot be compiled because of the ambiguous use of names. Hazardous side effects of macros are also well known.

Preprocessor and link seams together Preprocessor and link seams may be used together to mock a whole library: we can alter the include search path with specific preprocessor settings [56] and we can link with the test double implementation of the library. This technique has the disadvantages of both seams. Also, it is not possible to replace just a subset of a library. In case of the standard C or C++ libraries most often we want to replace only a subset of the library functions and not the whole library.

Object Seam

Object seams are realized by introducing a runtime interface. For example, let us consider the following C++ class (note, using a `const` qualifier on the `process()` member function and an RAII lock guard instead of explicit locking would make the code safer, but it would also make our message less visible):

```
// Entity.hpp
class Entity {
public:
  int process(int i) {
    if(m.try_lock()) {
      auto result = std::accumulate(v.begin(),v.end(),i);
      m.unlock();
      return result;
    }
    else { return -1; }
  }
  void add(int i) {
    m.lock();
    v.push_back(i);
    m.unlock();
  }
private:
  mutable std::mutex m;
  std::vector<int> v;
};
```

Figure 2.4: The 'Entity' class we would like to test

We would like to test the `Entity::process()` function for both possible return values of `try_lock`. Our objective is to have a test like this:

```
void test() {
  Entity e;
  set_try_lock_fails(e);
  ASSERT_EQUAL(e.process(1), -1);
  set_try_lock_succeeds(e);
  ASSERT_EQUAL(e.process(1), 1);
}
```

For the sake of the test, we introduce an interface and we can use it to change the behaviour of the mutex object in runtime. The production specific and test specific implementations of the interface need to be provided:

```
struct Mutex {
  virtual void lock() = 0;
  virtual void unlock() = 0;
  virtual bool try_lock() = 0;
  virtual ~Mutex() {}
};

struct RealMutex : Mutex { //used in production code
  void lock() override { m.lock(); }
  void unlock() override { m.unlock(); }
  bool try_lock() override { return m.try_lock(); }
private:
  std::mutex m;
};

struct StubMutex : Mutex { //used in test code
  // defintion of lock() and unlock() as before
  bool try_lock_result = false;
  bool try_lock() override {
    return try_lock_result;
  }
};
```

Now our class should use the interface:

```
class Entity {
public:
  Entity(Mutex& m) : m(m) {}
  int process(int i) { if(m.try_lock()) { ... } else { ... } }
  //...
private:
  Mutex& m;
  //...
};
```

We can see that the enabling point of this seam is the newly added constructor. The production and the test code might look like this:

```
void productionClient() {
  RealMutex m;
  Entity e(m);
  // some usage of e
}
void testClient() {
  // test setup
  StubMutex m;
  Entity e(m);
  // assertions ...
}
```

There is a severe problem with object seams that is illustrated via this specific example: the mutex object which was exclusively owned by the `Entity` now is moved outside. There is nothing to prevent any caller from reusing (misusing) this mutex. Regarding encapsulation this is fatal. Also, it is not clear who should create/destroy this object and when. The same problems arise if we replace the

`Mutex&` with a raw pointer or a smart pointer. Though passing a `unique_ptr` in the constructor and getting a reference to it via a getter/setter function might be an option, but then we would need to create that getter/setter function for `m` (to set up the test). Generally speaking, the following problems may arise when we replace dependency objects:

- Either we deprive the unit under test from the ownership of the dependency or we use a superfluous getter/setter function.
- We add an otherwise unnecessary constructor (or alternatively a setter function).
- We introduce superfluous pointer semantics via a reference or smart pointer, which is harmful to cache locality, hence it reduces overall performance [57].
- We have to introduce an interface just for testing. This interface has virtual functions. Calling them requires extra pointer indirections and this might result in cache misses and it loses the possibility of inlining, thus it harms the overall performance [58]. Adding an extra interface makes the program more complex, hence the program is harder to understand. Note that in some cases it might be possible to get rid of the additional explicit interface definition with type erasure [59], but the virtual function calls cannot be avoided even in this case.

Compile Seam

Our `Entity` and mutex example would be more natural if we make `Entity` a template and we use the `Mutex` type as a template parameter:

```
template <typename Mutex>
class Entity {
public:
  int process(int i) { if(m.try_lock()) { ... } else { ... } }
  //...
private:
  Mutex m;
  //...
};
```

However, because of testing we need to access the mutex outside of the `Entity` class. Therefore, one simple approach is to define a new getter/setter function:

```
template <typename Mutex>
class Entity {
public:
  int process(int i) { if(m.try_lock()) { ... } else { ... } }
  Mutex& getMutex() { return m; } // Use only from tests
  //...
private:
  Mutex m;
  //...
};
```

The enabling point of this seam is the template parameter itself. Client code may use our `Entity` with the appropriate type parameter:

```
void productionClient() {
  Entity<std::mutex> e;
  // some usage of e
}
void testClient() {
  struct StubMutex {
    //...
    bool try_lock_result = false;
    bool try_lock() {
      return try_lock_result;
    }
  };
  Entity<StubMutex> e;
  auto& m = e.getMutex();
  m.try_lock_result = false;
  ASSERT_EQUAL(e.process(1), -1);
  m.try_lock_result = true;
  ASSERT_EQUAL(e.process(1), 1);
}
```

We do not need to add an additional runtime interface this time, so the test client can use a `StubMutex` which does not have any virtual functions. Of course, the implicit compile-time interface [60, item 41] of `std::mutex` and `StubMutex` must match.

With this approach, we introduced a template parameter just because of testing. The original `Entity`, however, was perfectly natural to be a simple class, now it became a class template. Also, we added an extra getter/setter function to be able to drive the dependency externally from our class. Needless to say, we increased code complexity and compilation time [61]. There are some methods with which we could decrease compile time, but they would further complicate the source code (use of pimpl [62, item 22]) or the build system setup (using extern templates [63, 14.7.2]).

Link and preprocessor seams can be used to write non-intrusive tests. However, object and compile seams may be used for such purpose only if the unit under test already has the proper architecture. For instance, in case of object seams the unit must have a constructor (or setter) function to set up a different implementation

for the dependency. In case of compile seams, the unit must be a template and it must have a template parameter via which we can mock the dependency. Often, these architectural requirements are not satisfied, therefore the use of object and compile seams demand that we intrusively change the source code of the unit.

2.2.2 Test Automation Conventions

The test seams discussed above are frequently used in the process of creating automated tests. Thus, we briefly overview the general test automation patterns and we show the more specialized concepts about testing legacy code.

The *four-phase* test pattern is driven by the observation that each test requires some sort of setup and tear down routines. This pattern splits each test into four phases [64]. In the first phase, we set up everything that is required for the system under test (SUT) to exhibit the expected behaviour. In the second phase, we interact with the SUT. In the third phase, we do whatever is necessary to determine whether the expected outcome has been obtained. In the fourth phase, we tear down the test to put the world back into the state in which we found it. This pattern is also known as the build-operate-check-clear pattern [65].

The *given-when-then* pattern of representing tests is originated from behaviour-driven development [66, 67]. The *given* part describes the pre-conditions to the test. In these pre-conditions, we present the state of the world before we begin the behaviour we specify in the test. The *when* section represents the behaviour we specify. The *then* section describes the changes we expect due to the specified behaviour. We can also look at this pattern as a reformulation of the four-phase test pattern. Essentially these three states are equal to the first three states of the four-phase pattern.

In the context of the four-phase pattern, Robert C. Martin states that anyone who reads the tests should be able to work out what they do very quickly, without being misled or overwhelmed by details [68]. Consequently, both the four-phase and the given-when-then patterns imply that the test setup should be strictly part of the visible test code and should not be separated from the rest of the test code. For instance, using a link seam to set up a test separates the "given" phase from the rest of the test code, thus it violates both patterns and makes the test hard to understand.

2.3 New Non-intrusive Test Seams

We have shown in the previous section that all four existing C/C++ seams have some disadvantages that prevent us from using them or make us reluctant to use them for non-intrusive testing. Link seams do not work with inline functions and

require patching the build system. Link seams keep the test setup code separated from the other phases of the test. Thus, with link seams is not feasible to write non-intrusive tests which follow the given-when-then test pattern. Preprocessor seams are problematic with classes and namespaces. Object seams and compile seams often demand that we widen the public interface, thus they are intrusive. Also, object seams might introduce additional performance penalty in the production code.

In this section, we present new C/C++ test seams which may complement the existing seams and could be used for non-intrusive testing. We introduce three new non-intrusive testing seams:

1. a method based on compiler instrumentation and function call interception,
2. a procedure which transforms the original abstract syntax tree of the production code for testing,
3. a static reflection based approach.

The seam based on tree transformations is rather experimental because the underlying infrastructure is not stable enough. Also, the reflection-based seam is presented as possible future work, because compile-time reflection is not part of the core C++ language yet.

2.3.1 Function Call Interception based Test Seam

Function call interception (FCI) is one technique which enables non-intrusive testing by making it possible to replace function bodies. By replacing functions we can eliminate the unwanted dependencies in tests. With FCI we are able to intercept function calls at runtime and we can execute actions before and/or after the original function body or even completely replace it [69]. Different FCI methods have different advantages and disadvantages. Compared to languages like Java, C and C++ languages offer less mature solutions for FCI.

Function Call Interception Techniques

We differentiate the FCI techniques based on the time FCI is applied [69]. *Dynamic techniques* perform the interception at *program load-time* or at *runtime*. Contrary to dynamic approaches, *static techniques* achieve FCI by modifying the *source files* (e.g. with the help of the preprocessor), by changing the *linkage order*, by generating object files which contain the *instrumentation* or by modifying the application *binary image*; all these modifications happen before runtime.

Load-time FCI. Most modern operating systems provide the possibility to specify shared objects to be loaded before all others. This can be used to selectively override functions in other shared objects. E.g. on Linux this behaviour is controlled by the `LD_PRELOAD` environment variable [70]. With this technique, calling the original function is cumbersome. We have to use `dlsym` auxiliary function with the `RTLD_NEXT` argument [71]. In case of C++ functions, we have to provide the mangled names. Furthermore, this mechanism is unreliable with member functions (e.g. the member function pointer is not expected to have the same size as a void pointer on some platforms [72]).

Run-time FCI. In POSIX compliant systems, runtime dynamic interception is implemented with the help of the `ptrace` system call [73, 74]. If ptrace is used with the `PEEKTEXT` or `POKETEXT` argument then it is possible to attach to a running process and to read or write different segments of its memory. For instance, the GNU debugger (gdb) [75] and Intel Pin [76] both use this approach. A disadvantage of these tools is that they rely on a specific kernel functionality; thus porting these implementations to other operating systems may be hard. E.g. Intel Pin currently does not support function replacement on macOS [77]. Another property of this technique is that we cannot instrument inline functions.

Pre-compilation-time FCI. We consider some use of the C/C++ preprocessor as a pre-compilation-time interception. A typical use case is to replace the `malloc` and the `free` functions from the standard C library to collect statistics about the heap usage. This approach can be applied conveniently in C, but not in C++. As soon as we use namespaces, the preprocessor might generate code which cannot be compiled because of the ambiguous use of names. Hazardous side effects of macros are also well known.

Link-time FCI. One example for the link-time static interception is the `wrap` command line option of the GNU linker (`ld`) [78]. When this program option is applied then the linker uses a wrapper function for the specified `symbol`, any undefined reference to `symbol` will be resolved to `__wrap_symbol` and any undefined reference to `__real_symbol` will be resolved to `symbol`. This approach makes it possible to replace a function and call the original. However, in case of C++ we have to specify the mangled names as symbols. We cannot use this approach if the `symbol` is defined within the very same translation unit where it is referenced.

Post-compilation-time FCI. There exist tools to modify the compiled binary code for interception. As an example, in [79] the authors describe a method which is a mixture of Link-time and Post-compilation-time techniques used to avoid typical security vulnerabilities, like buffer overflow. A modified compiler can be applied on a binary executable (or shared library) to extract type information from the debugging data and reinsert it in the same binary which is then available at runtime in a special data structure. At runtime, a pre-loaded shared library

intercepts the possibly dangerous calls and validates them using the data structure stored in the first step.

Compile-time FCI. Perhaps the most widely used static FCI technique is to configure the compiler to emit instrumented code in a way that interception is possible. The GNU/GCC and LLVM/Clang compilers both provides the `-finstrument-function` program option to instrument each and every function call in a way to execute code before and after the body of the functions [80]. Actually, when this instrumentation is enabled then the compiler emits two extra calls for each function body. The prototypes of these two called functions are the following:

```
void __cyg_profile_func_enter(void *this_fn, void *call_site);
void __cyg_profile_func_exit(void *this_fn, void *call_site);
```

The arguments for these functions represent the address of the original function and the address of the instruction from where it was called. A serious limitation of this technique is that we cannot replace an intercepted function with another function; the original function will be called anyway.

Some seams are realized with FCI techniques. For instance, preprocessor seams are implemented with pre-compilation-time FCI. Link seams are realized with load-time and link-time FCI. The existence of compile-time, post-compile-time and run-time FCI drives us to further extend the list of existing seams. We define a new class of seams, the *FCI seams*. More specifically, we define three new seams for each FCI technique: *compile-time FCI seam*, *post-compile-time FCI seam* and *run-time FCI seam*.

In this dissertation we introduce a new compile-time FCI seam (2.3.1). We describe a run-time FCI seam in Appendix A which is based on the existing Intel Pin tool. However, we do not investigate post-compile-time FCI seams further.

Compile-time FCI Seam

In Figure 2.5 we present a legacy graphics program that relies on a LOGO-like API for drawing. The API is realized as a class named the `Turtle`. Also, there is `Painter` class which is responsible for drawing lines and shapes. This class has a hard-wired dependency on the concrete `Turtle` class. Still, we would like to write a test which checks the `DrawLine()` function. In this example let us suppose that the turtle functions are quite expensive to use. Generally speaking, a dependency may represent a database, or a network connection, whose usage can be hard, or very expensive. Therefore, in our test, we want to mock the `Turtle` class (or at least its member functions).

Our new instrumentation technique makes it possible to write non-intrusive tests easily. Figure 2.6 lists the test which uses our new instrumentation method.

```
// Turtle.hpp

class Turtle {
    int x = 0, y = 0;
public:
    void PenUp() { /* ... */ }
    void PenDown() { /* ... */ }
    void Forward(int distance) { /* ... */ }
    void Turn(int degrees) { /* ... */ }
    void GoTo(int x, int y) { /* ... */ }
    int GetX() const { return x; }
    int GetY() const { return y; }
};

class Painter {
    Turtle turtle;
public:
    void DrawLine(int x0, int y0, int x1, int y1) {
        turtle.GoTo(x0, y0);
        turtle.PenDown();
        turtle.GoTo(x1, y1);
        turtle.PenUp();
    }
    // ...
};
```

Figure 2.5: A legacy graphics program

We define our mock class (`MockTurtle`) with the help of the gmock macros (lines 6-10). Our test-case is defined from line 33 to 43. In the test-case we create an instance of the `Painter` class, then we get a reference to its private `turtle` member (line 37). Note that there are several different techniques to access a private member, we use a method which relies on explicit template instantiations [17]. Then we get a reference to an instance of the `MockTurtle` class which acts as a test double for the `Turtle` instance (line 38). We state our expectations as we would do with any other regular mock objects (lines 40-41). In line 42 we exercise our unit under test by calling the `DrawLine()` method. With the help of our tool we setup replacement functions for each member function of the `Turtle` class (lines 26-27). These replacement functions behave as a proxy; they forward each function call on a given `Turtle` instance to a corresponding test double (lines 17-22). The way we get the reference for a relevant test double is pretty simple in this test: we return a reference to a static instance of the `MockTurtle` class (lines 12-15). We can use this simplification because we know that there is only one `Turtle` object over the lifetime of our test-case. If there were several `Turtle` objects then we should solve the mapping differently, perhaps with the help of a static hash map. Lines 45-48 contains the definition for the `main()` function which

```
#include "Turtle.hpp"
#include <gmock/gmock.h>
#include <access_private.hpp>
#include <hook.hpp> // for SUBSTITUTE

class MockTurtle {
public:
  MOCK_METHOD0(PenUp, void());
  // PenDown, Forward, ...
};

MockTurtle& GetMockObject(Turtle*) {
    static MockTurtle m;
    return m;
}

namespace proxy {
    void PenUp(Turtle* self) {
        return GetMockObject(self).PenUp();
    }
    // Similarly to PenDown, Forward, ...
}

struct TurtleTest : ::testing::Test {
    TurtleTest() {
        SUBSTITUTE(Turtle::PenUp, proxy::PenUp);
        // Similarly to PenDown, Forward, ...
    }
};

ACCESS_PRIVATE_FIELD(Painter, Turtle, turtle)

TEST_F(TurtleTest, TestDrawLine) {
    using ::testing::AtLeast;

    Painter painter;
    Turtle& turtle = access_private::turtle(painter);
    MockTurtle& mockTurtle = GetMockObject(&turtle);

    EXPECT_CALL(mockTurtle, PenDown())
        .Times(AtLeast(1));
    painter.DrawLine(0, 0, 10, 10);
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Figure 2.6: Testing the legacy program with compile-time FCI

uses the functions and macros from googletest to initialize and run the test.

The most important property of this test is that the test setup is included in the test application itself. This test structure is enabled by our instrumentation technique which is controlled by the `SUBSTITUTE` macro. During the compilation of our test binary, we have to include a header file from our auxiliary runtime library which provides the `SUBSTITUTE` macro, and we have to enable the mentioned instrumentation with a compiler switch. Also, during linking, we have to link with our given runtime library.

As another example of this new approach, we present how to test the `Entity` class which has been introduced in Figure 2.4:

```
1  #include "Entity.hpp"
2
3  bool try_lock_result;
4  bool fake_mutex_try_lock(std::mutex* self) { return try_lock_result; }
5
6  TEST_F(Fixture, Mutex) {
7      SUBSTITUTE(&std::mutex::try_lock, &fake_mutex_try_lock);
8      Entity e;
9      try_lock_result = false;
10     EXPECT_EQ(e.process(1), -1);
11     try_lock_result = true;
12     EXPECT_EQ(e.process(1), 1);
13 }
```

Figure 2.7: Testing the 'Entity' class with compile-time FCI

With the `SUBSTITUTE` macro, we simply replace the `try_lock` member function of `std::mutex` with a free function named `fake_mutex_try_lock`. The first parameter of this free function holds a pointer to the object on which the original `try_lock` member function has been called. We set the desired return value of the free function via a global variable `try_lock_result`. In the test, once we set up the desired return value of the test double (line 9 and 11), then we exercise our unit under test and we formulate our expectations (line 10 and 12).

Our method has clear advantages compared to the `LD_PRELOAD` approach where we can substitute functions only if they are defined in shared libraries. With our technique, it is possible to write non-intrusive tests and replace even inline functions. However, this new method requires rebuilding the application (or unit) we want to test with the specific compiler option which will disable inlining. Our technique has the following advantages:

- The test setup is part of the test application and clearly visible together with the rest of the test code. This way it does not violate the given-when-then test automation pattern and best practices.

- It does not introduce a new tool into the existing build chain. The functionality is embedded into the compiler.
- On platforms where the compiler is supported, the new instrumentation could be supported as well.
- There is no need to use mangled names.
- We can use the ordinary unit test building tools and we can group unit tests into the same test application.

FCI with Call Expression Instrumentation

Our new interception technique and the prototype consists of two parts: a compiler instrumentation module and a runtime library. The instrumentation module modifies the code to check whether a function has to be replaced or not. The runtime library provides functions to set up the replacements.

Instrumentation During the code generation, we modify each and every function call expression to call an auxiliary function. Let us consider the following function call expression of `foo`:

```
foo(args...);
```

When our instrumentation is in action, the emitted code is equal to the following pseudo-code:

```
char* funptr = __fake_hook(&foo);
if (funptr) {
    funptr(args...);
} else {
    foo(args...);
}
```

The call to `__fake_hook` resolves at runtime if we should replace the callee with another function or not. We replace a function if the returned value of `__fake_hook` is not zero, in this case the returned value is a pointer to the function we call as a substitution. If the return type of the callee function is not `void` then we create an additional storage for the return value:

```
char* funptr = __fake_hook(&foo);
using ReturnType = decltype(foo(args...);
ReturnType ret;
if (funptr) {
    ret = funptr(args...);
} else {
    ret = foo(args...);
}
return ret;
```

Our prototype is based on LLVM/Clang [81, 82]. The LLVM compiler infrastructure is briefly presented in Appendix B. The implementation modifies the emitted LLVM Intermediate Representation (IR) [83] code. For instance, let us consider the following definition of the `bar` C++ function:

```
int foo(int);

int bar(int p) {
    return foo(p);
}
```

The LLVM IR of `bar` after optimization looks like this:

```
define i32 @_Z3bari(i32 %p) #0 {
entry:
  %call = tail call i32 @_Z3fooi(i32 %p)
  ret i32 %call
}
```

The generated code is very straightforward: there is only one basic block (`entry`) which stores the return value from the call of `foo` and then it returns with it. Note that the function names are mangled thus we see the `_Z3` prefix for the function names.

When we enable our instrumentation and optimization, then the IR has the form presented in Figure 2.8. Now we have four different basic blocks. The first block (`entry`) evaluates the return value[1] of the `__fake_hook` function, compares it to zero and emits a branch based on the comparison. The `then` block is executed if the callee shall be replaced. We call the substituting function pointer, then we jump to the last basic block(`cont`). The `else` block is executed if the callee shall not be substituted; we just simply call the original function then jump to the `cont` block. At last, in the `cont` block, we store the result of either the callee or the replaced function, and we return with that.

Clang's internal architecture [84] is built in such a way that the code generation for all kind of call expressions are eventually handled in one common routine. For

[1]The type of the return value is `i8*`, which is a pointer to an 8 bit long data and the most universal pointer type in LLVM. Thus, `char*` in C/C++ is the direct representation of `i8*` in LLVM. This is why we use `char*` instead of `void*` in our C/C++ code and in the pseudo codes above.

```
define i32 @_Z3bari(i32 %p) #0 {
entry:
  %fake_hook_result = tail call i8*
      @__fake_hook(i8* bitcast (i32 (i32)* @_Z3fooi to i8*))
  %0 = icmp eq i8* %fake_hook_result, null
  br i1 %0, label %else, label %then

then:                                             ; preds = %entry
  %1 = bitcast i8* %fake_hook_result to i32 (i32)*
  %subst_fun_result = tail call i32 %1(i32 %p)
  br label %cont

else:                                             ; preds = %entry
  %call = tail call i32 @_Z3fooi(i32 %p)
  br label %cont

cont:                                             ; preds = %else, %then
  %call_res.0 = phi i32 [ %subst_fun_result, %then ], [ %call, %else ]
  ret i32 %call_res.0
}
```

Figure 2.8: The LLVM IR when our instrumentation is enabled

example, in the case of virtual function calls the adjustment of the `this` pointer happens before calling that routine. We placed the emission of our instrumentation code inside that routine. As a result, special cases such as the `this` adjustment are automatically handled; we do not have to manually adjust the `this` pointer when we substitute a virtual function.

Contradictory to `-finstrument-functions`, by instrumenting the call expressions (and not the function body) we have the convenience that we do not have to recompile dependent libraries if the call expression is in code outside of the library. This has a clear advantage in case of system libraries, third-party shared libraries and security critical applications where we have to evade library interposing.

Runtime Library The main purpose of the runtime library is to implement the `__fake_hook` function which is referenced from the instrumented code. The realization of this hook function has to find the related function pointer in case of an active substitution. Essentially, it is a simple pointer to pointer mapping which may be implemented with a simple hash function. However, in order to make the lookup as fast as possible, we chose to implement the mapping with a simple offsetting into the virtual memory (shadow memory). During program startup – more precisely, when our shared object is loaded – we initialize the shadow memory with the help of the `mmap` [85] system call. We assume that the size of a function definition is at least 1 byte since it has to contain at least a return instruction. Let N denote the size of a pointer in bytes of a specific architecture. Since we must

be able to store a function pointer for every function, we have to reserve a shadow memory which is N times bigger than the normal virtual address space which holds the function definitions. If the `mmap` system call is called with the `MAP_ANONYMOUS` argument then it guarantees that the reserved memory is initialized to zero. Note that in practice the OS does not zero out the mapped region during the mapping, only at the moment when a virtual address is being accessed by the first time. We divide the user-space virtual memory into two different regions. Low memory and high memory. We handle the memory mapping differently for each region. For instance, on macOS the memory is partitioned as follows:

```
[0x7f0000000000, 0x7fffffffffff] || HighMem
[0x120000000000, 0x19ffffffffff] || HighShadow
[0x020000000000, 0x11ffffffffff] || LowShadow
[0x000000000000, 0x01ffffffffff] || LowMem
```

Let *addr* denote the original address, *shadowAddr* the address of the corresponding shadow and *shadowOffset* the offset for a region. With this formula $shadowAddr = addr * N + shadowOffset(region(addr))$ we can calculate the shadow address. By using the shadow memory instead of a simple hash map we trade execution time for space. The program occupies terabytes in virtual memory, however, the resident (physical) memory usage is equal to the number of used substitutions multiplied with N. More specifically, operating systems do not reserve the specific physical pages to the process until there is no write to that memory area. Consequently, those memory pages which contain the shadow values of substituted functions will be resident physical pages registered in the process page table. In practice, this means only a few kilobytes of additional physical memory usage when we run the tests (given a page has a 4kb size and not taking into account the Linux specific huge pages). During program startup, we must make sure that our shared object gets initialized before the first function call. Our prototype achieves this by setting the `constructor` attribute [86] on the initializer function of the shared object. If there are other shared libraries linked to the final executable with such initializer functions, then it is the user's responsibility to ensure that our library is initialized first.

Another purpose of the runtime library is to provide the user interface to set up the function substitutions. Replacing a function in C is pretty simple, the shared object defines a function for that:

```
_substitute_function((const char*)&foo, (const char*)&fake_foo);
```

We may use the `SUBSTITUTE` macro in case of C++ to replace functions; this construct is more generic because it also supports member functions. Note that we have to include the header file attached to the runtime library, also we have to link with it. Our implementation is thread-safe if there are multiple threads calling the very same function. Although, there is a race condition if one thread is

```
1  template <typename Class, typename MemPtr>
2  const char *address_of_virtual_fun(const Class *aClass, MemPtr memptr) {
3    const char **vtable = *(const char ***)aClass;
4    struct pointerToMember {
5      size_t pointerOrOffset;
6      ptrdiff_t thisAdjustment;
7    };
8    pointerToMember p;
9    memcpy(&p, &memptr, sizeof(p));
10   static const size_t pfnAdjustment = 1;
11   size_t offset = (p.pointerOrOffset - pfnAdjustment) / sizeof(char *);
12   return vtable[offset];
13 }
```

Figure 2.9: Get the address of a virtual function

calling the specified function while another thread is setting up the substitution; in such cases, the user code must ensure thread safety.

Virtual Functions A pointer-to-member function may have a different layout in case of virtual functions than in case of regular member functions. Therefore, we cannot just simply cast a virtual function pointer to a `void` pointer.

The naive approach Without compiler support, we can get the address of a virtual function in an architecture-dependent way. On Figure 2.9 we present how we can get the address in case of the Itanium C++ ABI [72]. First, we receive the vtable from an object by dereferencing its vpointer (line 3). The vpointer is the first element in the object. We interpret the bits of the pointer to member (`memptr`) as an instance of the aggregate class `pointerToMember` (lines 4-9). Next, we set up the architecture dependent function pointer adjustment (line 10). Then, we get the offset and return with the appropriate element in the vtable (lines 11-12). We could replace virtual functions by exploiting this technique. Let us suppose we have a macro named `SUBSTITUTE_VIRTUAL` which use this technique and the following class hierarchy:

```
struct B { virtual void foo(); }; struct D : B { void foo() override; };
```

If we wanted to replace the `foo()` function when the dynamic type was `D` then we would have to get a pointer to such an instance:

```
B* dummy = new D; SUBSTITUTE_VIRTUAL(&D::foo, dummy, &D_fake_foo);
```

However, to replace the function in the base class as well, we would have to get a pointer to an instance whose dynamic type was `B`:

```
B* dummy = new B; SUBSTITUTE_VIRTUAL(&B::foo, dummy, &B_fake_foo);
```

New compiler intrinsic The previous naive approach is ABI dependent and it also requires a reference to an existing object. Thus, we tried to find a better alternative without these restrictions. Generally speaking, in order to replace functions we just need an identifier for each function – virtual or not – which is unique in the program. Actually, each function has such a unique identifier, and it is its own address in the program's virtual memory. Unfortunately, there is no valid C++ language construct to get this unique identifier. Nevertheless, GCC has implemented this feature [87], but sadly Clang did not. Clang developers claim that this feature is fundamentally broken, because when we use it then the proper adjustment of the `this` pointer may be elided [88]. Still, our technique could use this feature since our compiler instrumentation intervenes after the `this` adjustment thunk is emitted. Thus, we implemented this functionality in the Clang compiler, so we are able to use it within our implementation, hidden from the users and enabled only in test code. With this approach, the replacement of the `foo()` function when the dynamic type is `D` has the following form:

```
SUBSTITUTE(D::foo, D_fake_foo);
```

This is the very same form which we can use to replace free functions or non-virtual member functions.

Internally, the `SUBSTITUTE` macro expands to a call to `_substitute_function` and the arguments of that function are generated by our new compiler intrinsic:

```
#define SUBSTITUTE(src, dst)                                          \
  do { _substitute_function((const char *)__function_id src,          \
                            (const char *)__function_id dst); } while (0)
```

We modified the compiler to parse a new kind of unary expression when the `__function_id` literal is given and the test specific instrumentation is enabled. In case of free functions and static member functions this unary expression has the very same type which we would get in case of the "address of" unary expression:

```
void foo();
void bar() {
  auto p = & foo; // void (*)()
  auto q = __function_id foo; // void (*)()
}
```

However, in case of non-static member functions the two expressions yield different types:

```
struct X { void foo(); virtual void bar(); };
void bar() {
  auto p = & X::foo; // void (X::*)()
  auto q = __function_id X::foo; // void (*)()
  auto r = __function_id X::bar; // void (*)()
}
```

At runtime the value of these expressions are evaluated to hold the address of the specific raw function which can be identified by the corresponding mangled name in the compiled binary's text section.

Overload Resolution We may have several functions with the same name but with different parameters. Let us consider the below code:

```
struct X {  int foo(int);  int foo(double);  };
int X_fake_foo_i(X*, int);
```

Normally, if we would like to get the address of `X::foo(int)` we have to explicitly cast a function pointer to the appropriate type:

```
int(X::*mfp)(int) = & X::foo;
```

Here, we define a pointer variable with the name `mfp` which has the type `int(X::*)(int)` and it holds the address of `X::foo`. With the `__function_id` intrinsic we have to do the same, but the type will be different:

```
int(*mfid)(int) = __function_id X::foo;
```

For safety reasons, the `__function _id` is hidden from the users of our instrumentation, but they can use the three parameter form of the provided `SUBSTITUTE` macro to replace an overloaded function. For example, to replace `X::foo` with the `X_fake_foo_i` free function one has to write:

```
SUBSTITUTE(int(int), X::foo, X_fake_foo_i);
```

Other Special Cases A few standard library functions, such as `abort` and `exit`, cannot return. Some programs define their own functions that never return. We can declare them `noreturn` to tell the compiler this fact The compiler can optimize without regard to what would happen if a `noreturn` function ever did return. This makes slightly better code [86]. Our prototype supports the substitution of functions with the `noreturn` attribute with functions which do return. We achieve this by generating such code for the call expression which we would generate in case of normal functions on the branch where the substitution is active.

Generally, during compilation, functions are not inlined unless optimization is specified. For functions declared inline, the `always_inline` attribute [86] inlines the function even if no optimization level was specified. Our implementation makes it possible to replace always-inline functions if a special program option is passed for the compiler. Naturally, the given function definition will not be inlined and

it will be emitted as a standalone function with an address. However, there are special cases with always-inline function declarations. For instance, in the case of the libcxx library – which is one standard C++ library implementation – most of the getters and setters have the `always_inline` attribute. For example, the `basic_string` class template declares `c_str()` as always inline but there is an `extern` template declaration in the `<string>` header for `basic_string<char>`. Also, there is an implicit template instantiation of `basic_string<char>` which does not expose `c_str()`, as that is declared to be always-inline. When we turn on our instrumentation, the generated object file has an undefined reference to `c_str()`, since the code is not emitted because of the `extern` template declaration. To make the instrumentation work either we have to recompile libcxx with our instrumentation enabled and we have to link against the instrumented libcxx, or we have to eliminate somehow the `extern` template declaration. The latter is possible in one of our branch of the libcxx repository. Note that we did not experience this interesting case with the GNU standard C++ library implementation on Linux (GCC/6.2 version).

A C++ `constexpr` function cannot be replaced when it is used in a compile-time expression. However, it can be replaced whenever it is used within a runtime context:

```
constexpr int foo(int p) { return p * p; }
int fake_foo(int p) { return p * p * p; }

TEST_F(FooFixture, Constexpr) {
    SUBSTITUTE(foo, fake_foo);

    // compile-time evaluation
    static_assert(foo(2) == 4, "");

    // runtime evaluation
    EXPECT_EQ(foo(2), 8);
}
```

The expression inside the `static_assert` is forced to be evaluated during the compilation.

Performance Evaluation

We measured the runtime performance of the compiled instrumented code with the help of the Adobe C++ Performance Benchmark [89]. This benchmark is used and accepted in the industry to measure the run-time performance of compiled (and instrumented) code [90, 91, 92]. A typical object-oriented program is characterized by its high level of abstraction. While this is normally an advantage for the development and maintainability of a program, it is a major headache for an optimizing compiler and for any compile-time instrumentation. By using

this benchmark we can measure the performance penalty caused by the instrumentation on the different abstraction levels like templates or function objects. We compare our instrumentation method to a non-instrumented debug build, to a non-instrumented release build and to a build when the `-finstrument-functions` instrumentation is enabled.

We use exactly three different test suites from the Adobe benchmark. Each of these test suites consists of a loop with each iteration sorting prepared data. The number of iterations and the size of the data is a reasonably big number to provide valuable measurements. The following pseudo-code describes the measurement:

```
void test_sort(Data Source, Data Dest)
{
  start_timer();
  for (int i = 0; i < iterations; ++i) {
    copy(Source, Dest);
    sort(Dest);
    verify_sorted(Dest);
  }
  record_result(timer());
}
```

The distinguished test suites measure overhead about different abstractions.

The function objects test suite The function objects test suite compares the performance of function pointers, functors, inline functors, standard functors, and native comparison operators. These function pointers and function objects are passed as a comparator to the sort algorithm. The sort algorithm is realized by a function template, which takes the comparator as a parameter:

```
template<class Iterator, typename Comparator>
void sort(Iterator begin, Iterator end, Comparator compare);
```

With this test suite, we measure the performance overhead caused by function pointers and function objects.

The Stepanov abstraction test suite The Stepanov abstraction test suite examines any change in performance when adding abstraction to simple data types. For instance, a value wrapped in a class may perform worse than a raw value:

```
template <typename T>
struct ValueWrapper {
  T value;
  ValueWrapper() {}
  template <typename TT>
  inline operator TT() const {
    return (TT)value;
  }
  template <typename TT>
  ValueWrapper(const TT& x)
      : value(x) {}
  T& operator*() const { return *value; }
};

template <typename T>
inline ValueWrapper<T> operator+(const ValueWrapper<T>& x,
                                 const ValueWrapper<T>& y) {
  return ValueWrapper<T>(x.value + y.value);
}

template <typename T>
inline bool operator<(const ValueWrapper<T>& x,
                      const ValueWrapper<T>& y) {
  return (x.value < y.value);
}
```

Also, a value recursively wrapped in a struct or class may perform worse than a raw value:

```
typedef ValueWrapper<double> DoubleValueWrapper;
typedef ValueWrapper<ValueWrapper<
    ValueWrapper<ValueWrapper<ValueWrapper<ValueWrapper<ValueWrapper<
        ValueWrapper<ValueWrapper<ValueWrapper<double>>>>>>>>>>
    DoubleValueWrapper10;
```

We measure the performance of an insertion sort when several layers of these value wrappers are applied.

The Stepanov vector test suite This test suite is similar to the previous one, but this time the abstraction is increased by moving from pointers to vector iterators. We measure the performance in case of applying quicksort on a `std::vector`. With this test suite, we can observe the overhead caused by iterators.

We compiled the test suites with different compiler flags and we compared the absolute total time of run time of each test cases. On Figure 2.10 we show the total time for the function objects test case with the different compiler setups. Our instrumentation is turned on with the `-fsanitize=mock` option. With the `-finstrument-functions` setup we define the `__cyg_profile` functions in a

standalone, separate translation unit. In case of both instrumentations, we disable inlining. Figure 2.11 presents the total absolute time for the Stepanov vector test suite. Similarly, Figure 2.12 represents the total time in one test case of the Stepanov abstraction suite.

Both our instrumentation and `-finstrument-functions` causes performance degradation. In most cases our technique performs similarly to `-finstrument-functions`. Our method exchanges two extra function calls (`__cyg_profile_func_enter` and `__cyg_profile_func_exit`) with one extra function call to `__fake_hook` followed by an efficient lookup.

We experienced that with our instrumentation, the size of the binary may grow bigger. In the case of a simple C program, we measured around 15% (bzip2). In the case of a template heavy program (Stepanov abstractions test suite), we measured that the executable could be up to 3 times bigger compared to a release optimized (`-O2`) binary. We measured very similar size growth in case of the `-finstrument-functions` feature.

We performed the measurements on a Linux machine with an Intel(R) Core(TM) i7-4610M CPU @ 3.00GHz processor and with 16GB RAM. The given CPU is laptop-class hardware that scales the frequency dynamically from 0.8Ghz to 3.7Ghz, therefore we turned off turbo boost and frequency scaling by using the appropriate ACPI kernel driver.

Note that we did not notice any degradation in the run time and in the memory usage of the compilation process itself when our instrumentation was turned on.

Limitations And Future Work

Our prototype is implemented in the code generation part of the Clang compiler, however, it would be architecturally better if we realized that as a transforming optimizer pass. This pass should run before all other optimizer passes. By having an optimizer pass, all the logic related to this instrumentation would be well separated and self-contained. Also, it would make it possible to use our tool with other language frontends, thus this is our most important future work. Currently, we do not have any check to enforce that the original function and its replacement have the same signature. In the future, we plan to create a checking function template for the substitutions. The prototype works only on 64 bit x86 systems.

Replace the `operator()` of a lambda is not supported unless we can take the address of the lambda. Similarly, member functions of `structs`/`classes` which are defined inside a function cannot be replaced, because there is no valid expression to get their address. Our technique relies on that we should be able to get the address of the function we want to substitute. In the case of constructors and destructors, we cannot get their address with any standard C++ expression. Still, replacing constructors or destructors would be a valuable contribution in the domain of

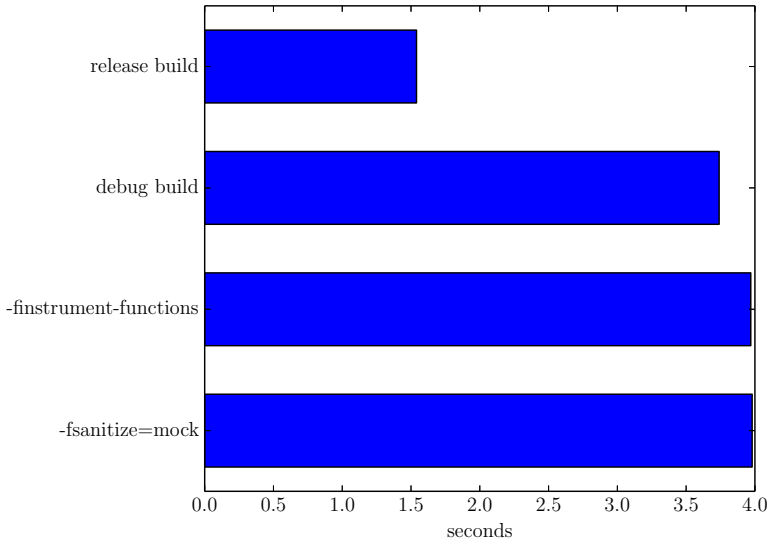


Figure 2.10: Total absolute time for function objects

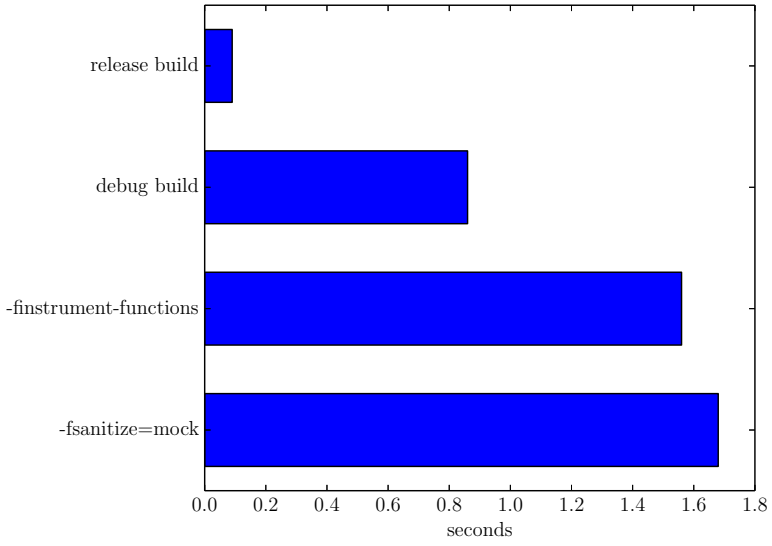


Figure 2.11: Total absolute time for vector quicksort

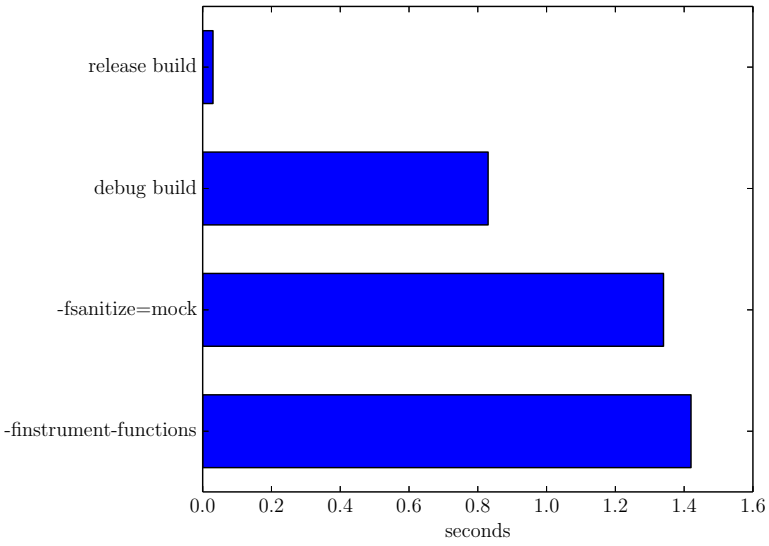


Figure 2.12: Total absolute time for abstraction insertion sort

testing, thus this is an important area for further research.

Related Work

Our new instrumentation technique for non-intrusive testing is based on compile-time FCI, which is a special function call interception method. The different function call interception techniques are explained in details by Kang [69]. The author also discusses aspect-oriented programming implementation techniques for intercepting method calls.

There are many existing software error checking tools which are based on some kind of instrumentation. These tools are worth to investigate because the instrumentation they apply could be used alternatively to the compile-time instrumentation which we applied in our work. A large number of memory error detectors are based on binary instrumentation. For example, Valgrind (Memcheck) [93], Dr. Memory [94], Purify [95], Intel Inspector [96]. The most popular compiler instrumentation based error checker tools are the AddressSanitizer [97], ThreadSanitizer [98], XRay instrumentation [99, 100] and other different sanitizers supported by the LLVM/Clang infrastructure [101, 102]. Our instrumentation technique was inspired by the AddressSanitizer, we reused many ideas from its implementation (e.g shadow memory).

Our new approach uses the so-called shadow memory to implement an ex-

tremely efficient hash table which contains the mapping between the functions and their respective test doubles. Shadow memory is not a new technique it has been used previously by various error checker software. The above mentioned AddressSanitizer and ThreadSanitizer both use shadow memory to store metadata for a specific piece of memory. AddressSanitizer uses a shadow space scaled down to one eight of the normal address space and can be easily used on 32 bit systems. However, ThreadSanitizer uses 8 times larger shadow memory than the normal address range, therefore support for 32-bit platforms is problematic and is not planned by the maintainers. We can find users of the shadow memory mapping outside of the LLVM universe as well. For instance, Valgrind [103] TaintTrace [104], LIFT [105], Bound-Less [106], Umbra [107, 108] and LBC [109].

Conclusion

Test seams are used to create non-intrusive tests for legacy systems, some of these seams are often realized via an FCI technique. We introduced our new compiler instrumentation for C and C++ programs, which makes it possible to replace the intercepted function call. While most of the existing instrumentation methods modify the function to call we instrument the caller side. We substitute the actual call with a small code snippet in compilation time, which decides at runtime whether the original or a replacement function is about to call. The decision is made using shadow memory and an offset to minimize runtime overhead. In contrast to other seams, our new instrumentation seam keeps the test setup code close to the other phases of the test. The technique makes it feasible to write non-intrusive tests which follow the given-when-then test pattern. This way, our method could help to implement well-structured tests for legacy software systems.

Compared to existing compile-time instrumentation solutions, our technique does not require the modification or even the recompilation of the intercepted functions, which is a possible advantage in case of legacy code, system libraries, third-party shared libraries or in situations when we have to avoid library interposing. We have created a prototype implementation using the LLVM/Clang compiler infrastructure, which is publicly available at [14].

2.3.2 Syntax Tree Transformation based Test Seam

FCI based test seams do not have the drawbacks of link and preprocessor seams. However, those techniques are still not applicable to all cases. For example, we cannot replace types, we can replace only simple function dependencies. Therefore, we seek for new seams, which do not have the disadvantages of link and preprocessor seams and enable us to replace dependent *types* with test double types.

In this section, we examine a new experimental non-intrusive testing approach which transforms certain parts of the original abstract syntax tree of the production code for the purpose of testing.

AST Transformations

An *abstract syntax tree* (AST) is a data structure which represents the hierarchical syntactic structure of the source program [110]. They are widely used in compilers as an intermediate representation of the program through several stages that the compiler requires. During certain compilation stages the abstract syntax tree is being transformed, e.g. in case of C++ template instantiation existing nodes are being modified and new nodes are added to the tree.

Abstract syntax tree transformations are used for several purposes. In the Groovy programming language [111], it is used as a form of compile-time metaprogramming; it allows the developers to hook into the compilation process and to modify the AST before bytecode generation.

The LLVM/Clang compiler infrastructure [112] makes it possible to write certain source to source transformations which modify the existing AST of a source file and produces the modified source code as an output. This may be used, for instance, to annotate the source code with additional statements that create statistics about memory allocations.

CodeBoost is a source-to-source transformation tool for domain-specific optimisation of C++ programs [113, 114]. CodeBoost performs parsing, semantic analysis and pretty-printing, and transformations can be implemented either in the Stratego program transformation language or as user-defined rewrite rules embedded within the C++ program.

Constant propagation, a well known data-flow optimization problem, can be implemented on abstract syntax trees in Stratego, a rewriting system extended with programmable rewriting strategies for the control over the application of rules and dynamic rewrite rules for the propagation of information [115].

Prolog can be retrofitted with concrete syntax and seamless interaction of concrete syntax fragments with an existing "legacy" meta-programming system (based on abstract syntax) can be achieved [116]. This can result in a considerable reduction of the code size and improved readability of the Prolog source code.

To support non-intrusive tests, our idea is to apply a special AST transformation. We would like to replace the AST node of the dependency with a new node which refers to the test double. However, in most cases, the AST contains cross-references to the nodes. For example, a call expression holds a cross-reference to the node of the callee. (Thus, strictly speaking, the AST is not a tree.) If we would like to replace a function definition node, then we have to replace all the declaration references in all call expressions. To achieve that we should have to

do a sophisticated traversal through the AST. Instead, we build the AST of the program starting with the test double nodes and then we build the rest of the AST of the production code on top of that. This way when we add a new node then the cross-references will point to the test double nodes. To achieve this incremental AST building we reuse and modify an existing algorithm: the AST import algorithm.

The AST Import Algorithm

Traditionally, C/C++ compilers process one *translation unit* (TU) at a time. However, there are use cases when we have to access information from several (or all) translation units of a program. For example, in case of link-time optimization, or in case of cross translation unit static analysis. The LLVM/Clang cross translation unit static analysis [117] executes the same single translation unit analysis algorithms but on an AST which is synthesized by merging several ASTs of the individual translation units. This merged AST is created by first parsing one TU and then importing the ASTs of the other TUs one-by-one into the first.

The algorithm to merge the ASTs is existing and implemented in the Clang compiler. It relies on the capability of detecting structural equivalence between different AST elements. Two AST nodes are *structurally equivalent* if they are

- builtin types and refer to the same type, e.g. `int` and `int` are structurally equivalent,
- function types and all their parameters have structurally equivalent types,
- record types and all their fields in order of their definition have the same identifier names and structurally equivalent types,
- variable or function declarations and they have the same identifier name and their types are structurally equivalent.

This description is depicted more formally in Algorithm 2.1. Note that templates and related AST nodes are handled similarly, but we do not include them in the algorithm, because we would like to keep it small and presentable.

Algorithm 2.2 describes the procedure of importing an AST. The algorithm has to ensure that the structurally equivalent nodes in the different translation units are not getting duplicated in the merged AST. E.g. if we include the definition of the vector template (`#include <vector>`) in two translation units, then their merged AST should have only one node which represents the template. Also, we have to discover *one definition rule* (ODR) violations. For instance, if there is a class definition with the same name in both translation units, but one of the definition contains a different number of fields. We refer to the translation unit

Algorithm 2.1: Structural Equivalence Check

```
function StructurallyEquivalentTypes(T1, T2)
    if T1 is Builtin and T2 is Builtin then
        if T1 == T2 then
            return True
    if T1 is Function and T2 is Function then
        forall (PT1, PT2) in (ParamTypes(T1), ParamTypes(T2)) do
            if ! StructurallyEquivalent(PT1, PT2) then
                return False
        return True
    if T1 is Record and T2 is Record then
        forall (N1, N2) in (FieldNames(T1), FieldNames(T2)) do
            if N1 ≠ N2 then
                return False
        forall (FT1, FT2) in (FieldTypes(T1), FieldTypes(T2)) do
            if ! StructurallyEquivalent(FT1, FT2) then
                return False
        return True

function StructurallyEquivalentDecls(D1, D2)
    N1 ← name(D1)
    N2 ← name(D2)
    if N1 ≠ N2 then
        return False
    T1 ← type(D1)
    T2 ← type(D2)
    return StructurallyEquivalentTypes(T1, T2)
```

into which we import as the *To* TU and the one which we import from as the *From* TU.

Algorithm 2.2: AST Import

```
for each top level Decl in the From TU do
    FoundDeclsList ← lookup all declarations in the To TU with the same
     name of Decl
    for each FoundDecl in FoundDeclsList do
        if StructurallyEquivalentDecls(FoundDecl, Decl) then
            ToDecl ← FoundDecl
            mark Decl as imported
        else if Decl is a function then overloaded function
            ToDecl ← create a new AST node in To TU
            import dependent declarations and types of ToDecl
        else
            report ODR violation
    if FoundDeclsList is empty then
        ToDecl ← create a new AST node in To TU
        import dependent declarations and types of ToDecl
```

Reusing the AST Import Algorithm for Testing

As for our contribution, we extend the already existing AST importer mechanism and we present how we can use that to create non-intrusive tests. With the extension, we can transmogrify the original AST of the production code to an AST which contains the nodes of test doubles instead of the nodes of the original dependencies. The key to our approach is that we import everything into the test double's context. When the production code is being merged and when we reach the import of a dependent entity then the lookup finds the definition of the test double and that definition is getting used in the rest of the imported AST. We introduced a new attribute (`[[test_double]]`) which modifies the behaviour of the structural equivalence check. During the check of two declarations/types, if this attribute is present in the declaration/type of the "to" context then we consider the two entity as equivalent. The modified structural equivalent check algorithm is presented in Algorithm 2.3. The structural equivalence check is called from the AST import algorithm where it is always guaranteed that the second parameter is the one from the "to" context. Thus, we check for the presence of the test double attribute for the second parameters ($T2, D2$). With the help of this new attribute and the modified equivalence check, we are able to change the import algorithm

Algorithm 2.3: Modified Structural Equivalence Check

```
function StructurallyEquivalentTypes(T1, T2)
    if T2.hasAttr(TestDoubleAttr) then
        return True
    if T1 is Builtin and T2 is Builtin then
        if T1 == T2 then
            return True
    ... Same as before in Algorithm 2.1

function StructurallyEquivalentDecls(D1, D2)
    if D2.hasAttr(TestDoubleAttr) then
        return True
    N1 ← name(D1)
    N2 ← name(D2)
    if N1 ≠ N2 then
        return False
    ... Same as before in Algorithm 2.1
```

to use the test double in the production code. We created a prototype implementation based on the LLVM compiler infrastructure which is publicly available at [15]. The LLVM compiler infrastructure is briefly presented in Appendix B.

Figure 2.13 demonstrates how our new method can be used to replace a simple function in the AST. The `foo()` function is defined in the `foo.c` translation unit and it calls the `bar()` function, which is defined in the very same TU. The `fake_bar.c` file contains the definition for the test double which we want to use as the replacement function in the test. The test double has the same name as the original function we want to replace. Normally, having two definitions would cause ODR violation during the import procedure, but this time the test double has the special attribute attached to its definition. In `test.c` we have the actual test code, which exercises the production code (`foo()`) and formulates expectations on that. If `foo()` calls the test double `bar()` then the return value will be `13` and then `main()` will return with success. First, we have to create the serialized AST files for each source file (`-emit-pch`). Then we import the AST of the production code (`foo.ast`) and the test code (`test.ast`) into the AST of the test double (`fake_bar.ast`). We achieve this by providing the AST files in the proper order and with the `-ast-merge` command line option of the compiler. Once we have the merged AST then we emit an object file from that (`-emit-obj`) and during this process, all C/C++ semantic checks are executed. Thus, if the modified AST is semantically incorrect then we will be notified. (Note that it would be possible

```
// foo.c
int bar() { return 1; }
int foo() {
  return bar();
}
```

```
// fake_bar.c
[[test_double]]
int bar() {
  return 13;
}
```

```
// test.c
int foo();
int main() {
  return
      foo() == 13 ? 0 : 1;
}
```

```
# Create the ASTs
clang -cc1 -emit-pch -o foo.ast foo.c
clang -cc1 -emit-pch -o fake_bar.ast fake_bar.c
clang -cc1 -emit-pch -o test.ast test.c

# Merge the ASTs and emit an object
clang -cc1 -ast-merge fake_bar.ast -ast-merge foo.ast -ast-merge test.ast /dev/null \
  -emit-obj -o merged.o
# Link
clang -o output merged.o

# Run the test
./output
```

Figure 2.13: Replacing a simple function in the AST

```
// foo.h
class Bar {
  int a;
public:
  int f() { return 1; }
};

class Foo {
  Bar bar;
public:
  int ff() {
    return bar.f();
  }
};
```

```
// fake_bar.h
class [[test_double]]
Bar {

public:
  int f() {
    return 13;
  }
};
```

```
// test.c
#include "foo.h"

int main() {
 return Foo().ff() == 13
             ? 0
             : 1;
}
```

Figure 2.14: Replacing a record type in the AST

to parse the source files on-demand instead of using the .ast files, however, that would result in a need for re-parse even if the related source did not change.)

In Figure 2.14 we show how we can replace a record type. This time we have a class as a dependency (`Bar`) defined in `foo.h`. The test double in `fake_bar.h` simply replaces the only `f()` member function, which returns a fake value. The rest of the mechanism is quite similar to before, the only difference is that this time we have to generate AST files from headers.

The previous example in Figure 2.14 presented that we can easily replace types if the substitute type has the same functions defined as the original one. Figure 2.15 demonstrates we can even extend the replaced type with additional functionality, which may be very useful if we want to create mock test doubles and not just simple stub objects. In the test (`test.c`) we would like to use a test double which

```
// foo.h
class Bar {
  int a;
public:
  int f() { return 1; }
};

class Foo {
public:
  Bar bar;
  int ff() { return bar.f(); }
};
```

```
// mock_bar_modifiers_fwd.h
struct Bar;
void set_f_return_value(Bar* bar, int value);
```

```
// mock_bar.h
#include "mock_bar_modifiers_fwd.h"

struct [[test_double]] Bar {
  int f_return_value = 0;
  int f() {
    return f_return_value;
  }
};

void set_f_return_value(Bar* bar,
                        int value) {
  bar->f_return_value = value;
}
```

```
// test.c
#include "foo.h"
#include "mock_bar_modifiers_fwd.h"

int main() {
  Foo foo;
  set_f_return_value(&foo.bar, 13);
  return foo.ff() == 13 ? 0 : 1;
}
```

Figure 2.15: Changing the AST to use a mock test double

can be set up to return with a given value in its member function (`f()`). To achieve this we provide a new header file (`mock_bar_modifiers_fwd.h`), which contains prototype definitions for such functions which can modify the dependency via a pointer. Since we use a pointer we do not have to see the whole definition of the `Bar` class, a forward declaration is sufficient. With this auxiliary header file we can create the AST dump for the test file. The definition of the setter function(s) is placed in the test double file (`mock_bar.h`) together with the definition of the mock type. The rest of the mechanism is similar to the previous examples, we have to merge the AST files of the production and test code into the AST of the test double. There is no need to create a separate AST file for the auxiliary header.

Evaluation

Compared to link seams, our new seam makes it possible to replace functions even if they are inlined or statically linked. Besides, we can replace class definitions and virtually anything because we do the replacement on the AST level. Similarly to link seams, our method requires additional help from the build system since we have to create the AST files separately and we have to merge them manually. Also, the enabling point of our seam is in the build scripts, as in the case of the link seams.

Similarly to the preprocessor seams, with our approach, we can replace not just functions but types or anything which has a name. However, our approach does the replacement on the AST level, while the preprocessor seam does that on the token level.

Limitations and Future Work

It is a common limitation with the four seams (link, preprocessor, FCI and ASTImporter) that we can replace only those dependencies which have an identifier name. E.g., it is not possible to replace a C++11 lambda function nested in a call of an algorithm from the standard template library:

```
std::vector<int> nums{1, 2, 3, 4, 5};
std::for_each(nums.begin(), nums.end(), [](int &n){ n++; });
```

As of this writing, the implementation of AST importer algorithm in LLVM/-Clang is quite immature. Especially, the import of C++ templates and C++11 expressions are not well supported. Thus, it is an important future work to improve this realization because that would open up the possibility to be able to experiment with our new technique on real C++ projects.

Conclusion

We presented a new non-intrusive testing approach which is based on transforming the AST of the production code in a way that dependent functions or types are replaced with test doubles. This method, although still experimental, has clear advantages compared to the previous FCI based solution because we can replace types as well, not just simple functions. Also, the technique exploits a compiler aided syntax tree modification which provides much safer transformation than we can get via tokeniser based translations with the preprocessor. Similarly to linker based solutions, our method requires additional help from the build system. A prototype implementation based on the LLVM/Clang compiler's ASTImporter module is publicly available at [15]. As the ASTImporter is still not stable at the time of writing this dissertation, we cannot complete a comprehensive evaluation of this approach.

2.3.3 Reflection based Testing and Test Seam

In this section, we present an idea for future research on non-intrusive seams. The idea is based on a reflection proposal which could be used to implement generic proxy objects and mock objects for unit testing frameworks. Although there is a long discussion of implementing compile-time reflection for C++, there is still no agreement on the topic. Therefore, our proposal is rather theoretical at the

moment, however, could be the basis of future research when there is consensus on C++ reflection.

Reflection Fundamentals

Reflection is the ability of a program to inspect or modify its own structure. In other words, reflection is referred to as the meta information associated with programming structures like types and functions. For example, in case of a class type, this meta information can provide the names and types of the class' fields. It is said that a code is doing *introspection* if it is observing its own state and structure. Also, when a code is capable of modifying its structure or state it is called *intercession*.

There are several uses of reflection. For instance, it is used for serializing objects, implementing language bindings, creating *object-relational mappings* (ORM) and implementing unit test frameworks with mock objects.

Compared to other mainstream programming languages, C++ is lagging behind in reflection. In this section, we analyze and summarize current C++ reflection capabilities and researches about compile-time reflection. Based on our analysis we introduce a new approach of compile-time reflection. This approach could be used to implement generic proxy objects and mock objects for unit test frameworks.

Compile-time and Runtime Reflection *Compile-time reflection* is about getting information which is internal to the compiler during the compilation process. Based on this information, the compiler's internal abstract syntax tree (AST) can be modified. Usually, this modification is not more than adding new nodes to the AST. This can be done either by normal language elements (i.e. by adding a new function) or by using some compiler intrinsics.

Runtime reflection is happening during the program's execution time. Usually, runtime reflection is implemented with the use of runtime metaobjects. This means there is a metaobject associated with each real object. During runtime, these metaobjects provide all the information and methods which are needed to achieve the reflection. These metaobjects are always part of the final executable, therefore making its size bigger. Runtime reflection works also with objects whose exact type is not known during compile time (i.e. dynamic polymorphic types).

Runtime reflection has a few drawbacks compared to the compile-time reflection: the executable's size will be bigger even if not all runtime objects are reflected, and our program will perform slower in runtime. There are languages where this is affordable, but in C++ the performance is a critical viewpoint, therefore runtime reflection is not a real option. On the other hand, compile-time reflection is not working with objects of dynamic polymorphic types.

Reflection in Other Languages Managed languages like Java and C# have a very strong and well-developed runtime reflection system.

In Java, it is possible to query a class' name, package info, superclass, implemented interfaces, methods, fields and annotations through the `Class` object. It is even achievable to get information about private members. Regarding methods, one can get all the parameters' type and the return type. It is also feasible to call one reflected member function (without knowing the exact name of it). Java reflection can be used to list an enum class' enumeration values as well. C# has similar reflection capabilities to Java [118, 119].

Scala provides both runtime and compile-time reflection. The compile-time reflection is realized in the form of macros, which provide the ability to execute methods that manipulate abstract syntax trees at compile time. Scala uses the so-called `Universe` to set up runtime or compile-time reflection. It is accomplishable to control the set of entities that we have reflective access to, by the so-called `mirror` [120].

The D programming language provides compile-time reflection through the `Traits` extension. It has very similar properties to the C++ type traits library, but a little bit more can be achieved with it [121].

Standardized Reflection in C++

RTTI Run-Time Type Information and `dynamic_cast` expressions can be used together to determine the dynamic type of an object of a polymorphic class. Under the hood, `dynamic_cast` might use similar or common implementation details to the `typeid` operator which results in a `type_info` object. Objects of class `type_info` can be compared, so the same polymorphic types will have the same objects. Since C++11, `hash_code` can be used which returns a value which is identical for the same types. Also since C++11 `type_index` is existing, which is a wrapper around a `type_info` object, that can be used as an index in associative and unordered associative containers [63]. RTTI can be considered as a runtime reflection in C++, however, the reflected metadata is simply not enough to execute higher level reflection tasks.

Type Traits Type traits are type related queries and type modifications, which can be executed during compile-time. Most of the queries are returning with a boolean value or with a simple integral value [63]. Examples:

- `is_integral` checks if a type is an integral type
- `is_same` checks if two types are the same
- `rank` obtains the number of dimensions of an array type

- `remove_reference` removes reference from the given type

Type traits are reflecting metadata of types, but with the help of them, it can only be decided whether a type has a specific property or not. Higher level reflection tasks, like querying names of all the fields of a class are impossible with them.

C++ Without Standardized Compile-Time Reflection

In this subsection, we discuss current C++ techniques which are widely used in industrial environments. We demonstrate through examples, why the life of a C++ programmer is harder without built-in compiler support for static reflection.

Serialization There is a boost serialization library which can be used both intrusively and non-intrusively [122]. The following example demonstrates the non-intrusive method (the original class is not modified):

```
struct gps_position
{
    int degrees;  int minutes; float seconds;
};
namespace boost { namespace serialization {
template<class Arch>
void serialize(Arch& ar,gps_position& g,const unsigned int ver) {
    ar & g.degrees;
    ar & g.minutes;
    ar & g.seconds;
} }} // namespace boost::serialization
```

We can see that for each and every new class a new template specialization have to be written. If there was static reflection, then serialization could be solved in a generic way.

Unit Test Mock Frameworks Figure 2.16 demonstrates an abstract class (`Turtle`) and its mock class. The mock can be used everywhere, where the original type appears as an interface. The mock class is created by Google's mocking framework, gmock [123]. It is an obvious drawback, that each and every function have to be defined by a macro. If the interface (the abstract class in this case) is a subject of change then the mock class has to be updated too. If there was static reflection then mock classes could be programmed in a generic way, and they could be created by the compiler.

```
class Turtle {
  virtual ~Turtle() {}
  virtual void PenUp() = 0;
  virtual void PenDown() = 0;
  virtual void Forward(int distance) = 0;
  virtual void Turn(int degrees) = 0;
  virtual void GoTo(int x, int y) = 0;
  virtual int GetX() const = 0;
  virtual int GetY() const = 0;
};
class MockTurtle : public Turtle {
 public:
  MOCK_METHOD0(PenUp, void());
  MOCK_METHOD0(PenDown, void());
  MOCK_METHOD1(Forward, void(int distance));
  MOCK_METHOD1(Turn, void(int degrees));
  MOCK_METHOD2(GoTo, void(int x, int y));
  MOCK_CONST_METHOD0(GetX, int());
  MOCK_CONST_METHOD0(GetY, int());
};
```

Figure 2.16: An abstract class and its mock class

Static Reflection There are workarounds for the missing static reflection. The below example demonstrates how to add metadata manually, without the compiler's help:

```
// Our existing struct
struct Foo { int i; bool j; /* ... */ };
// "Foo" as a Boost.Fusion sequence
BOOST_FUSION_ADAPT_STRUCT(Foo, (int, i) (bool, j))
struct Action {
    template<typename T>
    void operator()(T& t) const {
        // do whatever we need, e.g. serialize
    }
};
void usage() {
    Foo foo;
    boost::fusion::for_each(foo, Action{});
}
```

Here, the Boost.Fusion library is used [124], but there are several other similar libraries for this purpose. Someone, who is building a generic object-relational mapping library, might end up using something similar to this. Note that, when `Foo` is changing, the manually provided metadata have to be changed, so again one conceptual change requires at least two change in the editor.

Overview of C++ Reflection Proposals

The Reflection Study Group (SG7) of the ISO C++ Committee started its work at the fall of 2013 with the paper N3814 – Call for Compile-Time Reflection Proposals [125]. In this section, the most important compile-time reflection use cases are enumerated and the C++ community is asked to provide proposals to introduce compile-time reflection into the language. The use cases are:

1. Generation of common functions like equality operators, serialization functions. (Note that this implies the enumeration of class members.)
2. Type transformations like Struct-of-Arrays.
3. Compile-time context information (replacing assert).
4. Enumeration of other entities (namespaces, enums, etc).

Low-level intrinsics - N3815 In response to N3814, N3815 was written to give a proposal about the compile-time reflection of enumeration lists [126]. N3815 proposes to add three *Property Queries* to the Metaprogramming and Type Traits Standard Library that provide compile-time access to the enumerator-list of an enumeration type [126]. Specifically:

- `std::enumerator_list_size<E>`: the number of enumerator-definitions in the enumerator-list of `E`.
- `std::enumerator_identifier<E,I>`: the identifier from the `I`'th enumerator-definition.
- `std::enumerator_value<E,I>`: the value from the `I`'th enumerator-definition.

N4027 [127], N4113 [128] and N4428 [129] further extend the approach for classes. The reference implementation of these proposals [130] contains compiler-specific intrinsics, with which the above-mentioned Property Queries can be served. Amongst the enumerator related intrinsics there are other intrinsics implemented for querying member fields of a class:

- `record_member_field_count<A>`: the number of fields in `A`.
- `record_member_field_identifier<A, I>`: the identifier from the `I`'th field of `A`.
- `object_member_field_ref<A, a, I>`: the reference of the `I`'th field in object `a`, where `a` is an instance of `A`.

With the above three intrinsics, the *Generation of common functions* problem can be solved with recursive templates. Below we show a function which sums up all the members of a class:

```
struct A { int m_a; int m_b; int m_c; };

template <unsigned int Index>
struct Sum {
    static int f(A a) {
        return __object_member_field_ref(A,a,Index) +
            Sum<Index - 1>::f(a);
    }
};
template <>
struct Sum<0> {
    static int f(A a) {
        return __object_member_field_ref(A,a,0);
    }
};

int summa(A a)
{
    return Sum<__record_member_field_count(A) - 1>::f(a);
}
```

We can create a common generic summa if we make `A` to be a template parameter of the `summa` function. Note that instead of the template recursion we could use `make_index_sequence<>` [63] together with a variadic template and with a parameter pack expansion.

Though other reflection proposals may have a more user-friendly interface, the low-level intrinsic approach provides simple intrinsics for each well-defined fundamental reflection task. More complex queries and reflection functionalities can be built gradually on top of the fundamental reflection elements. For instance, the reference implementation can be easily extended to query the number of methods [16].

Static Reflection via Template Pack Expansion N3951 (and later P0255R0) proposes to gather the metadata at once, without a "size + index" interface [131, 132]. From a type `T`, obtain static typed reflection adding two language constructs:
(1) An instruction `typename<T>...` that expands members identifiers of type `T` into a variadic template. Each type of n-th element of `typename<T>...` is a `const char*` and each n-th value is the identifier of n-th member of `T`, expressed in UTF-8 encoded;
(2) An instruction `typedef<T>...` that expands members of type `T` into a variadic template (in the same order of `typename<T>...`). Each n-th type of `typedef<T>...` is the type of the n-th member of `T` and each n-th value is a

pointer to n-th member of T, or a value if member is a `constexpr` member or enum item; `typename<T>...` and `typedef<T>...` could be implemented in terms of the N3815 related lower level "size + index" reflection traits as a library.

Reflection with Concepts In P0385R1, Matus Chochlik, Axel Naumann and David Sankel use a `reflexpr` operator to associate a unique implementation-defined class with each reflected type (fundamental, compound, user-defined), namespace, and specifier (public, virtual, etc.) [133]. A set of queries, in the form of type traits, is used to access the name, members, and other properties of the reflected class. Certain queries (i.e., `get_pointer<>`) can be used to access the reflected objects or class members.

Exposing the AST There are proposals which aim to solve reflection related tasks with a completely different aspect, for instance, N3883 [134] and P0590R0 [135]. They try to answer this question: How to solve enumeration of members without template recursion? They would like to avoid template metaprogramming in case of reflection tasks. Also, the goal of these proposals is to expose an AST like interface into the language with which all the metadata can be queried.

Compile-time Strings Compile-time strings play an important role as being the carrier of a reflected identifier's name. N3815 and N3951 propose to use `char` arrays as a carrier for names, this is because currently there is no better alternative in C++. However, it might be possible that in the future a `basic_string_literal` will be introduced as it is stated in D3933 [136].

Code Generators Code generators like Qt's Meta Object Compiler (MOC) [137] and OpenC++ Meta Object Protocol (MOP) [138] extends the base C++ language with some reflection and metaobject creation capabilities. This approach does not modify the C++ language, instead, a pre-compile phase needs to be added to the compilation process. Before the C++ compiler is called, the meta compiler must be invoked to translate the extended C++ into standardized C++. We can see the obvious disadvantages:

1. One additional compilation step is needed along with a new parsing and semantic analysis.

2. Lack of standardization.

The goal of SG7 is to provide a powerful native reflection, with which such pre-compilation is not needed.

Static Reflection for Proxy and Mock Objects

In the following, we describe our reflection approach which could help to create a generic proxy or mock object. First, we describe proxy and mock objects, then we present our proposal.

Mock and Proxy Objects (and Classes) *Mock objects* are used in unit tests to substitute real dependencies of a unit. (A unit is typically a class (struct) or a free function.) The programmer can formulate expectations towards a mock object, e.g. how many times a member function is called with a certain value? *Proxy objects* are those objects which have the exact same interface as the original object, but the implementation of each member function could be different. Therefore, mock objects are a special kind of proxy objects. Proxy objects seemed to be so useful that Java introduced the *Dynamic Proxy* concept to ease the creation of proxies [139]. Mock objects are instances of *mock classes*, proxy objects are instances of *proxy classes*.

A *simple aggregate class* is a C++ struct with publicly available fields and without methods. The definition of a *simple aggregate proxy class* is a recursive definition: A simple aggregate class is a proxy class if all of its fields have a proxy class type. The built-in types like `int`, `double`, `float` are considered as proxy types.

Proposed Approach - New Declarations Our proposal further extends the low-level intrinsic approach. We choose to extend the low-level intrinsics because more complex queries and reflection functionalities can be built gradually on top of the fundamental reflection elements; there is no need to implement first a very heavy reflection operator like `reflexpr` in P0385R1.

To successfully solve the problem of creating proxy and mock classes we have to support two new declarations.

1. `variable_decl` for declaring and defining variables based on reflected types and names.

2. `function_decl` for declaring and defining functions based on reflected types and names.

These declarations ideally would be mapped under std::reflect namespace. This mapping is needed in order to hide the compiler specific implementation details. This is the exact case with some already existent type traits as well, e.g. `std::is_pod`.

Declare a New Variable Let's assume we have the following simple struct:

```
struct A {
    int m_a;
    float m_b;
};
```

Then we can define a new class which has the same field as `A`:

```
struct B {
    reflect::variable_decl<
        reflect::record_member_field_type<A, 0>,
        reflect::record_member_field_identifier<A, 0> >;
};
```

This is equivalent to:

```
struct B {
    int m_a;
};
```

In this example, `variable_decl` has two subexpressions

1. A *type-specifier*, which refers to the newly declared variable's type.
2. A *compile-time string*, which is equal to the **name** of the newly declared variable.

The type-specifier is an expression whose value is a type, which can be evaluated during the compilation process. For instance, this can be a result of any kind of meta function or can be a result of any kind of reflection expression. Note that the new declaration `variable_decl` is not bound to any concrete reflection query implementation. It just requires the first parameter to be a type. The compile-time string can be either the C++14's compile-time string which is a simple `char` array; or this can be a *basic_string_literal* as described in D3933 proposal [136]. We use `record_member_field_identifier`, as it is implemented in the N3815 proposal related implementation. We introduce and use a new expression `record_member_field_type<T,N>` which is equal to the type of `T`'s `N`-th field type.

The declaration `variable_decl` should be handled as a normal variable declaration/definition. Thus, we can initialize a variable like this:

```
struct B {
    reflect::variable_decl<
        reflect::record_member_field_type<A, 0>,
        reflect::record_member_field_identifier<A, 0> > = 0;
        //                  initialization expression: ^^^^
};
```

Once `variable_decl` is implemented then an aggregate proxy class can be created recursively for `struct A`:

```
template <unsigned int Index> // C has the same field names as A,
struct C : C<Index-1> {       // but all fields are proxied.
    reflect::variable_decl<
        Proxy<reflect::record_member_field_type<A, Index>>,
        reflect::record_member_field_identifier<A, Index> >;
};
template <>
struct C<0> {
    reflect::variable_decl<
        Proxy<reflect::record_member_field_type<A, 0>>,
        reflect::record_member_field_identifier<A, 0> >;
};
```

Note that the start of the recursion is missing, that will be elaborated later. Here we assume such a `Proxy` class is existent, which can do the proxying for all field types of `struct A`.

If the proxy task is mocking (being able to create expectations), then we assume it is possible to create a `Proxy` class for each primary types (integral type, floating point type, pointer type, etc) and POD types.

By making `struct A` to be a template parameter we get the generic proxy aggregate struct:

```
template <typename A, unsigned int Index>
struct D : D<A, Index-1> {
    reflect::variable_decl<
        Proxy<reflect::record_member_field_type<A, Index>>,
        reflect::record_member_field_identifier<A, Index> >;
};
template <typename A>
struct D<A, 0> {
    reflect::variable_decl<
        Proxy<reflect::record_member_field_type<A, 0>>,
        reflect::record_member_field_identifier<A, 0> >;
};
```

The template recursion must be started with the number of fields in type `A`:

```
template <typename A>
struct GenericAggregateProxy :
    D<A, reflect::record_member_field_count<A>> {};
```

`record_member_field_count` is implemented in N3815's reference implementation [130]. The declaration `variable_decl` should be implemented similarly as type traits expressions. In the case of Clang this means

- A new token should be introduced in TokenKinds.def.
- Parsing actions should be created in `clang::Parser`.
- Semantic analysis should be added to `clang::Sema`.
- A new AST node should be introduced for `variable_decl`.

- Template instantiation rules for this AST should be given.[140]

The template instantiation rules should include the template transformation rules, which finally should result in a modified AST for this new declaration. The template transformation rules should be delegated back to the original `clang::FieldDecl` AST transformations. Similarly, intermediate code generation could be delegated as well.

Declare a New Method On the way to provide a generic mock class, the next step is to be able to declare and then to define functions based on reflected information. That is the exact purpose of introducing the `function_decl` declaration. The idea is similar to the one in case of `variable_decl`. Let us assume that all member functions in `struct A` have only one parameter. The following recursively built `struct C` has exactly the same functions *declared* as `struct A`:

```
struct A {
    int m_func1(int);
    float m_func2(float);
};
template <unsigned int Index> // C has the same functions as A
struct C : C<Index-1> {       // but they all have one parameter
    reflect::function_decl<
        reflect::record_member_function_result_type<A, Index>,
        reflect::record_member_function_identifier<A, Index>,
        reflect::record_member_function_param<A, Index, 0> >;
};
template <>
struct C<0> {
    reflect::function_decl<
        reflect::record_member_function_result_type<A, 0>,
        reflect::record_member_function_identifier<A, 0>,
        reflect::record_member_function_param<A, 0, 0> >;
};
```

In this case, the start of the recursion will use the number of member functions in `struct A`. Here `function_decl` has three subexpressions:
(1) A *type-specifier*, which refers to the newly declared function's return type.
(2) A *compile-time string*, equal to the **name** of the newly declared variable.
(3) The *parameter type* of the function. In this case, the declared function has only one parameter.

We introduce `record_member_function_result_type`, `record_member_function_identifier` and `record_member_function_param` as new compile-time expressions which evaluate to the return type, name and parameter type of the n-th member function. In case of functions with multiple parameters, we introduce `record_member_function_param` as a type list:

```
// C has exactly the same functions as A
template <unsigned int Index>
struct C : C<Index-1> {
    reflect::function_decl<
        reflect::record_member_function_result_type<A, Index>,
        reflect::record_member_function_identifier<A, Index>,
        // list of types !
        reflect::record_member_function_params<A, Index> >;
};
template <>
struct C<0> {
    reflect::function_decl<
        reflect::record_member_function_result_type<A, 0>,
        reflect::record_member_function_identifier<A, 0>,
        reflect::record_member_function_params<A, 0> >;
};
```

Definition of functions based on reflected information is more complex:

```
template <unsigned int Index>
struct C : C<Index-1> {
    reflect::function_decl<
        reflect::record_member_function_result_type<A, Index>,
        reflect::record_member_function_identifier<A, Index>,
        reflect::record_member_function_params<A, Index> >
    {
        struct Handler {
            template <typename... Ts>
            auto operator()(std::tuple<Ts...>& args)
            {
                // ...
            }
        };
        Handler{}(reflect::function_decl_params);
    }
};
template <>
struct C<0> { /* ... similar as before */ };
```

We add `reflect::function_decl_params` again as a new expression, which would be exposed as an `std::tuple` object. Each n-th type of the tuple should be the type of the n-th function parameter, and each n-th value shall be a reference to the n-th function parameter. Note that it might be more feasible to use a function parameter pack instead of `std::tuple`, but for the ease of explanation, we used the tuple.

Reflection Seam

By building a higher-level reflection library based on our proposed approach it would be possible to create non-intrusive tests:

```
class Entity {
public:
  int process(int i) { if(m.try_lock()) { ... } else { ... } }
  //...
private:
  std::mutex m;
  //...
};

void testClient() {
  using EntityUnderTest =
    test::ReplaceMemberType<Entity, std::mutex, StubMutex>;
  EntityUnderTest e;
  auto& m = e.get<StubMutex>();
  // Test code as before
}
```

Here, `EntityUnderTest` is a type alias to such a type, which is equivalent to the `Entity` type except that all of its members with type `std::mutex` are replaced by the `StubMutex` type. Also, this type could give access to its internal mutex instance via its `get` function template. Actually, `ReplaceMemberType` is a special *proxy class* template, which could be built similarly as we built the generic mock class template previously. For this technique to work, the given class has to be header-only because the compiler has to know its internal layout and types to be able to replace some of them with another type. This requirement might be harsh, but upcoming C++ modules [141] may mitigate this disadvantage.

Evaluation

Our reflection proposal is based on the current reflection proposal (low-level intrinsics), which provides a quite clear interface for the most primitive cases. However, in order to realize a generally usable reflection seam which covers the full language we have to answer some open questions. For instance, regarding the variables:

1. How to declare static variables?
2. How to handle C++14's templated variables?

In respect of the functions:

1. How to handle template functions?
2. How to handle constructors?
3. How to handle ellipsis function parameters?
4. How to handle exception specifications?

Before answering the above questions, first, we must decide how to reflect ellipsis, template functions, exception specifications, etc.

Conclusion

Reflection in C++ is a hot research area and it is a subject of frequent changes. Many application areas require it, but the approaches to define a firm interface are different. Reflection itself is a large topic, it is not even clear what meta information could be queried in future C++. Despite of these uncertainties, it is sure that the most general reflection queries like getting the fields of a class will be part of some future C++ standard. We presented an approach with which declaring or defining new variables and functions based on reflected meta information is possible. With this approach, it is possible to create generic classes which could behave as generic proxy, mock or serialization classes.

2.3.4 Contribution

Thesis 1 (New non-intrusive testing methods)**.** *I have analysed the existing dependency replacement techniques of C++ for testing and evaluated their advantages and disadvantages. I have introduced and analysed three new non-intrusive testing approaches: (1) I have implemented a method based on compiler instrumentation and function call interception. The new technique has clear advantages, thus it provides an alternative way to replace dependencies. I have created and evaluated a prototype implementation which is publicly available. (2) I have presented an experimental procedure which transforms the original abstract syntax tree of the production code for testing. With this procedure, it is possible to replace not just simple functions but also types. I have created a proof-of-concept prototype to demonstrate that the idea is feasible. (3) I have proposed a static reflection based approach as a future direction. Besides replacing types this solution could be used to implement generic proxy and mock objects for unit test frameworks.*

thesis name	relevant publications									
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
(1) **New non-intrusive testing methods**	•	•	•	•					•	
(2) Extending access for non-intrusive and white-box testing	◦							◦		◦
(3) Selective friend					◦					
(4) High-level abstraction for the read-copy-update pattern						◦	◦			

2.4 Comparison of Existing and New Seams

In this section, we enumerate the arguments for and against each existing and the newly introduced non-intrusive seams. Object and compile seams are not listed here because in most of the cases they cannot be used as a non-intrusive seam (for details please refer to 2.2.1). We include in the list, however, the Intel Pin based non-intrusive run-time FCI seam which is described in appendix A. The various advantages are depicted with the + sign, while the disadvantages with the − sign. Figure 2.17 summarizes the strengths and weaknesses of the listed non-intrusive seams.

Link Seam

+ Available and stable implementation on all platforms.
+ No runtime performance overhead.
+ There is no need to recompile the production code neither the dependent libraries.
+ Can replace functions in shared libraries.
− C++ support is limited, we have to use mangled names.
− The test setup code is separated from the other phases of the test.
− Cannot replace inline functions.
− Cannot replace function templates.
− Cannot replace functions if they are in static libraries or in the same translation unit.
− Cannot replace types.

Preprocessor Seam

+ Available and stable implementation on all platforms.
+ No runtime performance overhead.
+ The test setup code is not separated from the other phases of the test.
+ Can replace inline functions.
+ Can replace function templates.

+ Can replace functions if they are in static libraries or in the same translation unit.
+ Can replace types.
+ There is no need to recompile the dependent libraries, only the production code has to be recompiled.
+ Can replace functions in shared libraries.
− The production code must be recompiled.
− C++ support is very limited: namespaces are not supported, macros have hazardous side effects.
− Cannot replace anything consistently in C++ because the lack of handling namespaces.

Run-time FCI based Seam

+ Stable implementation.
+ There is no need to recompile the production code unless we want to replace inline functions.
+ Can replace functions in shared libraries.
+ Can replace functions in static libraries or in the same translation unit.
− Not available on all platforms. E.g. on macOS we cannot replace functions.
− There is a runtime performance overhead: 2x - 90x [142].
− C++ support is limited, we have to use mangled names.
− The test setup code is separated from the other phases of the test.
− Cannot replace inline functions. (We have to recompile with inlining disabled.)
− Cannot replace function templates.
− Cannot replace types.
− We need a completely new tool (Pin) to be introduced into the existing build chain. Developers and maintainers of the project must learn and understand Pin.

– The setup or tear-down of different test case may require clearing the substitutions of the functions. There is no obvious way to clear all the function replacements, therefore the easiest way is to have a new Pin tool for each test case. To launch a new process for each test cases may decrease the run time of the test suite compared to the case where all test cases are in one process.

Compile-time FCI based Seam (2.3.1)

+ Can replace functions in shared libraries.
+ Can replace functions in static libraries or in the same translation unit.
+ Available on all platforms (where LLVM/Clang is available).
+ Extensive C++ support (constructor and destructor cannot be replaced).
+ The test setup code is not separated from the other phases of the test.
+ Can replace inline functions.
+ Can replace function templates.
+ There is no need to recompile the dependent libraries, only the production code has to be recompiled (because of call expression instrumentation).
– Prototype implementation.
– The production code has to be recompiled.
– Cannot replace types.
– There is a runtime performance overhead: 2x - 20x.

AST Transformation based Seam (2.3.2)

+ No runtime performance overhead.
+ Can replace functions in shared libraries.
+ C++ support is unlimited.
+ Can replace inline functions.
+ Can replace function templates.

+ Can replace functions if they are in static libraries or in the same translation unit.
+ Can replace types.
– Only a proof-of-concept prototype implementation is available.
– It is needed to recompile the production code and the dependent libraries to generate PCH files.
– The test setup code is separated from the other phases of the test.

Compile-time Reflection based Seam (2.3.3)

+ No runtime performance overhead.
+ There is no need to recompile the dependent libraries.
+ Can replace (member)functions in shared libraries.
+ C++ support is unlimited.
+ The test setup code is not separated from the other phases of the test.
+ Can replace inline (member)functions.
+ Can replace (member)function templates.
+ Can replace (member)functions if they are in static libraries or in the same translation unit.
+ Can replace types.
– Only a design plan is available, but there is no existing implementation.
– It is needed to recompile the production code.
– It is not possible to replace free functions (we can generate new free functions only based on the reflected metadata).
– There is no actual replacement of functions or types, instead, we always have to generate a new type for the test which contains the replaced member functions or types.

Properties	Existing Seams			New Seams		
	Link Seam	Pre-processor Seam	Run-time FCI based Seam (Intel Pin)	Compile-time FCI based Seam	AST Transformation based Seam	Compile-time Reflection based Seam
Replace function	Yes, in shared libs.	Yes.	Yes.	Yes.	Yes.	Yes, but member functions only.
Replace type	No.	Yes.	No.	No.	Yes.	Yes, but member types only.
Replace in same TU or in static lib	No.	Yes.	Yes.	Yes.	Yes.	Yes.
Replace in shared lib	Yes.	Yes.	Yes.	Yes.	Yes.	Yes.
Replace inline or template function	No.	Yes.	No.	Yes.	Yes.	Yes.
Place of the test setup	Build system.	In the test code.	In the Pin tool.	In the test code.	Build system.	In the test code.
C++ support	Limited, with mangled names.	Limited. Problem with namespaces and macros.	Limited, with mangled names.	Extensive.	Full.	Full, standard proposals are under discussion.
Is recompilation needed?	No.	Yes.	No.	Yes.	Yes.	Yes.
Place of instrumentation	N/A.	N/A.	Function.	Call expression.	N/A.	N/A.
Stability of the implementation	Stable.	Stable.	Stable.	Prototype.	Proof-of-concept prototype.	No implementation available.
Supported platforms	All.	All.	Not all, macOS is not supported.	All.	All.	All.
Runtime performance overhead	None.	None.	2x - 90x. (See [142])	2x - 20x.	None.	None.

Figure 2.17: Comparison of existing and new seams

2.5 Access Private Members

Non-intrusive testing may require the ability to access private data. Imagine a situation where a replaced member function has to access the internal state of the object to be able to formulate an assertion on that. For instance, when testing the `Entity` class with our FCI method (Figure 2.7), we may wish to check that the `mutex` is unlocked when the `try_lock` member function is called. Thus, the test double `fake_mutex_try_lock` has to access private data:

```
bool fake_mutex_try_lock(std::mutex* self) {
  EXPECT_EQ(self->locked, true);
}

TEST_F(FooFixture, Mutex) {
    SUBSTITUTE(&std::mutex::try_lock, &fake_mutex_try_lock);
    // ... as before
}
```

When we access private data during testing we actually do white-box testing. White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) is a method of testing software that tests internal structures or workings of an application [143, 144].

In this section, we present exiting methods for accessing private members and we overview their strengths and weaknesses. Later we introduce our new generic and non-intrusive library. Currently, this library is the only available approach that makes it possible to access private members without violating the C++ standard. Then we present our other solution which is based on a prototype implementation of a language extension which makes it possible to access private members by out-of-class defined friend functions.

2.5.1 Existing Methods

There are various existing methods available for accessing private members in C++, but all have certain drawbacks. In this section, we overview the existing approaches and we introduce two new procedures that attempt to overcome the difficulties.

Access via a shared reference or pointer

In this case, the unit that we wish to test has a constructor or a setter function with a reference or a pointer parameter through which we can inject the dependency. An example is:

```
Entity(Mutex& m) : m(m) {}
```

We cannot use a `unique_ptr` this way since we need to access the dependency from the test as well; however, `unique_ptr` provides exclusive ownership only for the owner.

Access via getter

As we saw previously, it might be more natural to use a getter when the unit has exclusive ownership:

```
// Use only in tests !
Mutex& getMutex() { return m; }
```

The disadvantage here is that it violates encapsulation, i.e. it exposes an internal member.

Use preprocessor and getter

To protect the internal member, it is possible to define the getter function only when the unit is built for testing.

```
#ifdef TEST
  Mutex& getMutex() { return m; }
#endif
```

This requires support from the build system, so during the compilation of each translation unit of the test executable this flag needs to be defined. Also, test specific preprocessing is hardcoded, which makes it more difficult to see through the unit's overall structure. Though it is quite obvious that the getter is used only for testing, this is actually a benefit.

Change the access level with the preprocessor

A frequently applied trick is to use the preprocessor to access private members:

```
#define private public
#include "Unit.h"
#undef private
// Test code comes from here
```

Although the standard forbids us from redefining keywords [63, 17.6.4.3.1 Macro names/2], most preprocessors accept this. The obvious drawbacks are easy to see, however. All the other classes that are directly or indirectly included from `Unit.h` now expose all their internals. This opens the possibility for errors in the test code via accidentally accessing members of the dependencies, which we shall not know about (violating encapsulation).

Also, this approach does not always work. To see this, consider the following `class`:

```
class X { int a; };
```

The default access specifier is `private` in the case of C++ classes, therefore the define directive has no effect. Still, this can be circumvented with an additional define: `#define class struct`.

What is more, this is undefined behaviour because the C++ standard specifies that the order of allocation of non-static data members with different access control is unspecified [63, 9.2 Class members/13].

Friend function or class

In C++, friends have the right to access private and protected members. With this approach, we declare a concrete test function inside the unit to be a friend:

```
class Entity {
public:
  friend void testClient(Entity& e);
  Entity(std::unique_ptr<Mutex> m) : m(std::move(m)) {}
  int process(int i) { if(m->try_lock()) { ... } else { ... } }
  //...
private:
  std::unique_ptr<Mutex> m;
  //...
};
void testClient(Entity& e) {
  // access e.m here
}
```

We can also declare an in-between class to be a friend; then in the tests, we can use the different member functions of the friend class to access the private members. Declaring a test friend is still an intrusive act, which changes the internal source code of the unit which we want to test. Sometimes we cannot or do not want to modify a class like that. The reasons behind that might be the following:

- It is part of a third party software package and
  - Our build system would overwrite the changes we made
  - We do not want to maintain our own version
- Touching the internals would require a tremendous amount of recompilation of client code, which might not be desired.

Thus, we seek for such non-intrusive techniques which do not require changes in the unit we wish to test.

2.5.2 Access via Explicit Instantiation

So far we have considered only intrusive methods to access private members. In this section, we present an interesting non-intrusive technique then we present our new solution for accessing private members as a generalization of this technique.

We can access outside of the declaring class any private member if we exploit the fact that C++ allows us to pass the address of a private member in explicit instantiation [63, 14.7.2 Explicit instantiation/12]. The standard permits this behaviour because otherwise specializing traits for private types would not be possible. Besides private members, we can access private `static` variables and functions as well with this technique.

To understand how we can exploit this fact, consider the following class:

```
class A { static int i; };
int A::i = 42;
```

We would like to access the static private variable `i`. Normally, accessing that private variable results in a compiler error:

```
int x = A::i; // Error, i is private
```

Yet, there is an exceptional case, namely when we provide a template argument in an explicit template specialization. Let us assume that we have a class template defined, so we can explicitly specialize that:

```
template struct private_access<&A::i>;
```

The template argument `&A::i` has a compile-time available value and it has the type `int*`. In this context, `&A::i` is a completely valid expression, which has the address of the private variable as the value. We need to expose this address somehow, so we define the class template `private_access` as follows:

```
template <int* PtrValue> struct private_access {
  friend int* get() { return PtrValue; }
};
```

The template parameter of `private_access` is a non-type template parameter, which is a pointer value of type `int*` known at compile-time. We define the `get()` function to return the actual compile-time value of this template parameter. It returns the address of the private static variable since the template is instantiated with that value as an argument. By defining the `get()` function as a friend it becomes part of the enclosing namespace scope. Even so, its name is not found by normal lookup (qualified or unqualified) [63, 7.3.1.2 Namespace member definitions/3]. Therefore, we need to provide an additional *declaration* outside of the class:

```
int* get();
```

Putting this all together, our code with a usage example is the following:

```
class A { static int i; };
int A::i = 42;

template <int* PtrValue> struct private_access {
  friend int* get() { return PtrValue; }
};

int* get();

template struct private_access<&A::i>;

void usage() {
  int* i = get();
  assert(*i == 42);
}
```

The access of a private, non-static member is quite similar:

```
1  class A { int i = 42; };
2
3  template<int A::* PtrValue> struct private_access {
4      friend int A::* get() { return PtrValue; }
5  };
6
7  int A::* get();
8
9  template struct private_access<&A::i>;
10
11 void usage() {
12     A a;
13     int A::* ip = get();
14     int& i = a.*ip;
15     assert(i == 42);
16 }
```

The only difference is in the type of the template argument, which is now `int A::*`, a pointer to member. Values of this type may be pointers to any `int` data member of the class `A`. Once we get the pointer to the member in line 13, we can bind this pointer to an object, and this way, we get a reference to the data member (in line 14).

Generalized private access

As for our contribution, we generalized the above-presented techniques. We have created a library which automates the generation of the helper constructs to access private data members and to call private member functions (both static and non-static) [17]. Currently, this library is the only available approach which is generic and non-intrusive and makes it possible to access private members without violating the C++ standard.

Accessing private data members becomes straightforward with the library:

```
class A { int m_i = 3; };

ACCESS_PRIVATE_FIELD(A, int, m_i)

void foo() {
  A a;
  auto &i = access_private::m_i(a);
  assert(i == 3);
}
```

Similarly, calling private functions can be achieved like so:

```
class A {
  int m_f(int p) { return 14 * p; }
};

ACCESS_PRIVATE_FUN(A, int(int), m_f)

void foo() {
  A a;
  int p = 3;
  auto res = call_private::m_f(a, p);
  assert(res == 42);
}
```

We deliberately do not use pointer-to-members in the public interface of this macro library. We think that their use is just an implementation detail that we do not wish to expose to the user.

As a first design decision, we place all components of this library into an unnamed namespace to prevent multiple definition linker errors. For instance, we want the following 3 files (`a.hpp`, `x.cpp`, `y.cpp`) to be linkable into an executable file:

```
// a.hpp
class A { int m_i = 3; };


// x.cpp
#include "A.hpp"
#include "access_private.hpp"
ACCESS_PRIVATE_FIELD(A, int, m_i)


// y.cpp
#include "A.hpp"
#include "access_private.hpp"
ACCESS_PRIVATE_FIELD(A, int, m_i)
int main() { return 0; }
```

Then we commence with the generic definition of `private_access`. We use the nested namespace `private_access_detail` as a safeguard because we wish to

avoid name clashing as the user code might have additional names defined in an unnamed namespace:

```
namespace {
  namespace private_access_detail {
    template <typename PtrType, PtrType PtrValue, typename TagType>
    struct private_access {
      friend PtrType get(TagType) { return PtrValue; }
    };
  } // namespace private_access_detail
} // namespace
```

By introducing the `PtrType` type template parameter, we generalize the type of the pointer we wish to use. This might be `int*` or `int A::*` if we take our examples from the previous section. We also bring in the `TagType` type template parameter, which we use to define different instances of the `get()` function. This is achieved implicitly by instantiating the `private_access` class template with different concrete tag types.

Next, we define some helper macros for concatenation:

```
#define PRIVATE_ACCESS_DETAIL_CONCATENATE_IMPL(x, y) x##y
#define PRIVATE_ACCESS_DETAIL_CONCATENATE(x, y)                    \
  PRIVATE_ACCESS_DETAIL_CONCATENATE_IMPL(x, y)
```

We use the `PRIVATE_ACCESS_DETAIL` prefix for all the implementation macros that are supposed to be hidden from the clients of this macro library.

Afterwards, we introduce a macro which contains all those things that are common in the implementation of accessing a static or a non-static member:

```
1  #define PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE(Tag, Class, Type, Name, \
2                                               PtrTypeKind)            \
3    namespace {                                                        \
4      namespace private_access_detail {                                \
5        struct Tag {};                                                 \
6        /* Explicit instantiation */                                   \
7        template struct private_access<decltype(&Class::Name),         \
8                                       &Class::Name, Tag>;             \
9        /* Define the PtrType alias */                                 \
10       using PRIVATE_ACCESS_DETAIL_CONCATENATE(Alias_, Tag) = Type;   \
11       using PRIVATE_ACCESS_DETAIL_CONCATENATE(PtrType_, Tag) =       \
12           PRIVATE_ACCESS_DETAIL_CONCATENATE(Alias_,                  \
13                                             Tag) PtrTypeKind;        \
14       /* declare the get() function */                               \
15       PRIVATE_ACCESS_DETAIL_CONCATENATE(PtrType_, Tag) get(Tag);     \
16     }                                                                \
17   }
```

The macro parameter `Tag` is the name of the tag class we want to define and we also use it as a suffix for the name of the type aliases. `Class` denotes the qualified or unqualified name of the class we wish to provide access to. `Type` is the type of the

member variable. The parameter `PtrTypeKind` describes what kind of pointer are we dealing with, namely a simple pointer or a pointer-to-member. For instance, it may have the strings `*` or `A::*`. First, we define the tag type (line 5), then comes the explicit instantiation with the type and address of the member and with the recently defined tag type (line 7-8).

Then, we define a type alias (with `PtrType_` prefix) for the concrete type of the pointer (line 9-13). Basically, this alias is formed from the concatenation of the `Type` and `PtrTypeKind` parameters. For example, in the case of a pointer-to-member, the canonical type of the type alias might be `int A::*`. The twist here is that we need to add two type aliases because pointer-to-member-functions cannot be expressed generically with one alias, e.g.:

```
using PtrType1 = int(int) *; // ERROR
using Alias = int(int);
using PtrType2 = Alias *; // OK
```

Next, we declare the `get()` function to make it available for finding by normal name lookup (line 15).

Following this, we define the specific macro for non-static member fields.

```
1  #define PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE_FIELD(Tag, Class, Type, \
2                                                     Name)             \
3    PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE(Tag, Class, Type, Name,       \
4                                         Class::*)                     \
5    namespace {                                                        \
6      namespace access_private {                                       \
7        Type &Name(Class &&t) {                                        \
8          return t.*get(private_access_detail::Tag{});                 \
9        }                                                              \
10       Type &Name(Class &t) {                                         \
11         return t.*get(private_access_detail::Tag{});                 \
12       }                                                              \
13       using PRIVATE_ACCESS_DETAIL_CONCATENATE(X, Tag) = Type;        \
14       using PRIVATE_ACCESS_DETAIL_CONCATENATE(Y, Tag) =              \
15           const PRIVATE_ACCESS_DETAIL_CONCATENATE(X, Tag);           \
16       PRIVATE_ACCESS_DETAIL_CONCATENATE(Y, Tag) &                    \
17           Name(const Class &t) {                                     \
18         return t.*get(private_access_detail::Tag{});                 \
19       }                                                              \
20     }                                                                \
21   }
```

The macro parameters `Tag`, `Class`, `Type` and `Name` have the exact same meanings as before. In line 3, we call the previously described macro to generate all the generic code we need. We pass "Class::*" as a macro argument since we are dealing with non-static members. If we were dealing with static members, then the argument would be "*". Then, we define two overloaded functions in the enclosing `access_private` namespace with the name which is equal to the name of the

member we are exposing, e.g. "m_i" (line 7-12). These overloads are for those cases where the class instance is bound to an rvalue reference or to a non-const lvalue reference. We bind the result of `get()` function to the object of the class, and then we return with a reference to the resulting member. Later, (in lines 13-19) we add another overload that handles the cases where the object is bound to a const lvalue reference. In this case, we would like to preserve the constness of the object, therefore we should return with a const reference to the member. So, we create a type alias for this const reference type (lines 13-15). Here, once again we need to use two type aliases because we would like to avoid warnings that arise from duplicated const qualifiers. If we used just one alias, then we would get this warning if the `Type` macro parameter already contains a const qualifier. After defining the type alias, we use this in the definition of the third overload (lines 16-19).

The implementation of accessing static fields is very similar to the implementation of accessing non-static members.

The realization of calling private member functions, however, requires an explanation:

```
1  #define PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE_FUN(Tag, Class, Type,   \
2                                                   Name)               \
3    PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE(Tag, Class, Type, Name,       \
4                                         Class::*)                     \
5    namespace {                                                        \
6      namespace call_private {                                         \
7        template <typename Obj,                                        \
8                  std::enable_if_t<std::is_same<                       \
9                      std::remove_reference_t<Obj>, Class>::value> * = \
10                     nullptr,                                         \
11                 typename... Args>                                    \
12       auto Name(Obj &&o, Args &&... args) -> decltype(               \
13           (std::forward<Obj>(o).*get(private_access_detail::Tag{}))( \
14               std::forward<Args>(args)...)) {                        \
15         return (std::forward<Obj>(o).*                               \
16                 get(private_access_detail::Tag{}))(                  \
17             std::forward<Args>(args)...);                            \
18       }                                                              \
19     }                                                                \
20   }
```

Here, we again call the common macro that does the explicit instantiation (lines 3-4). Then we perfect forward both the object and the parameters of the private function we wish to call (lines 7-17). We bind the pointer-to-member-function (the result of the `get()` function) to the perfect forwarded object and then we call the resulting member function with the forwarded arguments (lines 15-17). We use the same expression's type as the trailing return type in the header of the function (lines 12-14). (Note that in C++14 there is no need to specify the trailing return type.) We also restrict the set of function template instantiations

that can participate in the overload resolution with the `enable_if`. The goal here is to exclude a template function when the type of the object is different from the type of the `Class` parameter. By doing this, we get a more compact error message if we misuse the library for some reason. Otherwise, we would get error messages originating from the body of the function template.

The implementation of calling static member functions is very similar to the implementation of calling non-statics, but we do not need the `enable_if` there since we do not have an object in that case.

Somehow we need to generate unique tag types, so for this, we use the built-in `__COUNTER__` macro which returns an integer and is incremented by the preprocessor each time it is referenced. `__COUNTER__` is not a standard macro, but it is available on most mainstream compilers (GCC, Clang, MSVC).

```
#define PRIVATE_ACCESS_DETAIL_UNIQUE_TAG                                \
  PRIVATE_ACCESS_DETAIL_CONCATENATE(PrivateAccessTag, __COUNTER__)
```

The macro `PRIVATE_ACCESS_DETAIL_UNIQUE_TAG()` will generate a unique name with the prefix `PrivateAccessTag`. Finally, we can define the main macros of the library with the help of the unique tag generator:

```
#define ACCESS_PRIVATE_FIELD(Class, Type, Name)                         \
  PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE_FIELD(                           \
      PRIVATE_ACCESS_DETAIL_UNIQUE_TAG, Class, Type, Name)

#define ACCESS_PRIVATE_FUN(Class, Type, Name)                           \
  PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE_FUN(                             \
      PRIVATE_ACCESS_DETAIL_UNIQUE_TAG, Class, Type, Name)
```

During the compilation of one translation unit, each invocation of these macros generates different tag types. Since these tag types are defined in an unnamed namespace, we will not have any linkage errors of duplicate symbols when linking multiple translation units together.

Now we have seen how we can access private member fields and how we can call private member functions, regardless of whether if they are static or not. However, this library has some limitations. We cannot access private types because the only valid context of using that private type is inside the explicit instantiation. We cannot call private constructors nor destructors. This is because a pointer-to-member cannot bind to a constructor (since we do not have the object unless the constructor is called). Nor can a pointer-to-member bind to a destructor because there is no valid expression in C++ to grab the address of a destructor. We have a link time error in the case of in-class declared `const static` variables (without an out-of-class definition). This is because we would take the address of that variable, and if that is not defined (i.e the compiler does a compile-time insert of the const value), we would be trying to dereference an undefined symbol. Owing to all of these limitations we were motivated to come up with a more sophisticated solution.

Note that the Java language has a built-in support to achieve something similar. With `setAccessible` we can indicate that the reflected object should suppress access checking when it is used [145].

2.5.3 Out of Class Friend

As we saw above, private access via explicit instantiation does not work for all kinds of private entities. So, our other contribution is to explore the idea of a new lingual element with which we would be able to access all kinds of private members. In our non-intrusive approach, we define a function or a class as a `friend` out of the befriending class:

```
template <typename Mutex>
class Entity {
public:
  int process(int i) { if(m.try_lock()) {} else {} }
  //...
private:
  Mutex m;
  //...
};

friend for(Entity<StubMutex>) void test_try_lock_fails() {
  Entity<StubMutex> e;
  auto& m = e.m; // access the private member
  // set up try_lock result value to false and do the assertions ...
}
```

Based on the LLVM/Clang[2] compiler (version 3.6.0) [82], we created a proof-of-concept implementation for out-of-class friends and it is now available online (see [18]). The goal of this implementation is to demonstrate that the idea is indeed feasible, though it is not our objective to provide a full-featured perfect realization. Therefore, we add some restrictions to the functionality and we do not implement proper error handling.

To ease the implementation, we use C++ attributes [63, 7.6 Attributes] instead of modifying the existing grammar. More specifically, we use the GCC `__attribute__` syntax because the standard `[[attribute]]` syntax implementation was not complete in the Clang version we used. By using attributes, we skip the problem of parsing and we can focus on the new semantic actions. So, the above definition of `test_try_lock_fails` with attributes is the following:

```
__attribute__((friend(Entity<StubMutex>)))
void test_try_lock_fails() {
  //...
}
```

[2]The LLVM compiler infrastructure is briefly presented in Appendix B.

However, prior to this definition, we need to explicitly instantiate the `Entity` class template.

```
template class Entity<StubMutex>;
```

This is required because the attribute's associated semantic action attempts to access all the details of its type parameter (`Entity<StubMutex>`). In a future study, it might be possible to modify the realization so as to implicitly do the instantiation during the semantic action of the friend attribute. The instantiation could be triggered just before accessing the details of the type parameter.

The definition with the attribute behaves exactly like any other in-class defined friend definition. As such, it is not found by normal lookup unless we declare it explicitly. Of course we wish it to be found by normal lookup, otherwise, we will not be able to call the function. Overall, this means that our test code should have the form:

```
template class Entity<StubMutex>;

__attribute__((friend(Entity<StubMutex>)))
void test_try_lock_fails() {
  //...
}
// explicit declaration makes it available for normal lookup
void test_try_lock_fails();

// part of the test framework
void testDriver() {
  test_try_lock_fails();
  // ... call other test functions
}
```

Here `testDriver` is the function which embodies the test framework, whose task is to execute each test case (or test suite) one-by-one. Another restriction of this particular realization of out-of-class friends is to allow only functions to be declared friends in this way.

After defining the constraints of such an attribute-based implementation we can explore the concrete realization steps. First, we define our new attribute in Clang's `Attr.td` file:

```
def OutOfClassFriend : InheritableAttr {
  let Spellings = [GCC<"friend">];
  let Args = [TypeArgument<"Host">];
  let Subjects = SubjectList<[Function]>;
  let Documentation = [Undocumented];
}
```

`Spellings` defines the list of the supported attribute syntaxes, but this time it is only the GCC style. The attribute syntax also defines the name of the attribute, in our case it is `friend`. `Args` specifies the list of the attribute arguments. Our

friend attribute has only one argument which is a type. This type argument refers to the type that we would like to be the host class (the befriending class), i.e. the class for which we define the additional friend function. `Subobjects` describe the list of the lingual elements that might have this attribute. In this case, we only allow functions to have it. Note that implementing this attribute for classes is an important issue for future research.

Once we have the attribution definition in place, the Clang infrastructure will generate all the necessary parsing code. What is left is for us to define the semantic action for the new attribute and to hook that action into the existing compiler machinery. As for the hooking, we need to add a new function call in the `ProcessDeclAttribute` function. This function is dedicated to apply a specific attribute to the specified declaration if the attribute applies to declarations. (Our attribute applies to function declarations.)

```
static void ProcessDeclAttribute(Sema &S, Scope *scope, Decl *D,
                                  const AttributeList &Attr,
                                  bool IncludeCXX11Attributes) {
  //...
  case AttributeList::AT_OutOfClassFriend:
    handleOutOfClassFriendAttr(S, D, Attr);
    break;
  //...
}
```

The semantic action for the new attribute is defined in Figure 2.18. The `S` parameter holds a reference of the monumental `Sema` class which is responsible for semantic analysis and AST building in the Clang compiler. The `D` parameter represents the declaration which has the attribute. The attribute itself is described with the `Attr` parameter. The first step is to get the type parameter of the attribute as a `QualType` (line 3-12). A `QualType` holds the basic type (e.g. `int`) and all the qualifiers – if any – on that type. For this, we get the `ParsedType` from the `Attr` with the `getTypeArg()` function (lines 4-6). A `ParsedType` is an opaque pointer for `QualType`s, i.e. this is a type-erased generic holder, this is something similar to `void*`. Next, we get the underlying `QualType` from the `ParsedType` and we set the location of it (lines 10-12). If we cannot get the location information for the type, we simply set it to the location of the attribute (lines 11-12).

Afterwards, we get the `RecordDecl` instance from the `QualType` instance with the help of the `getRecordDecl` function (line 15). This function returns a null pointer if the `QualType` does not represent a record declaration. In Clang, a `RecordDecl` is the type of the AST node that is created for C structs and unions. Similarly, a `CXXRecordDecl` is specifically for C++ classes, structs and unions. This means that we can safely cast the record declaration to a `CXXRecordDecl` (line 17). The cast expression used here is a Clang specific cast, which is a "checked cast" operation. It converts a pointer or reference from a base class to a derived

```
1  static void handleOutOfClassFriendAttr(Sema &S, Decl *D,
2                                         const AttributeList &Attr) {
3    // Get the attribute type argument as QualType
4    ParsedType PT;
5    if (Attr.hasParsedType())
6      PT = Attr.getTypeArg();
7    else { // TODO error handling
8    }
9    TypeSourceInfo *QTLoc = nullptr;
10   QualType QT = S.GetTypeFromParser(PT, &QTLoc);
11   if (!QTLoc)
12     QTLoc = S.Context.getTrivialTypeSourceInfo(QT, Attr.getLoc());
13
14   // The type argument must be a CXXRecordDecl
15   RecordDecl *RD = getRecordDecl(QT);
16   assert(RD);
17   CXXRecordDecl *CRD = cast<CXXRecordDecl>(RD);
18   // The attribute is subject of a FunctionDecl
19   FunctionDecl *FD = cast<FunctionDecl>(D);
20   // Set this function as a friend function
21   FD->setObjectOfFriendDecl();
22   // Create a new friend decl for the befriending class
23   FriendDecl::Create(S.Context, CRD, D->getLocation(),
24                      cast<NamedDecl>(D), Attr.getLoc());
25   // For the record, Add the attribute to the Decl
26   D->addAttr(::new (S.Context) OutOfClassFriendAttr(
27       Attr.getRange(), S.Context, QTLoc,
28       Attr.getAttributeSpellingListIndex()));
29 }
```

Figure 2.18: Semantic action for the out-of-class friend attribute

class, causing an assertion failure if it is not really an instance of the right type [146].

Next, we get the more specific function declaration (`FunctionDecl`) from the parameter `Decl` (line 19). The conversion from `Decl` to `FunctionDecl` must succeed since we explicitly specified in the `Attr.td` file that this attribute is valid only for function declarations. So we use the checked cast again. Then we set up this function declaration as a friend declaration (line 21).

Later, we create the AST node for this new friend declaration (lines 23-24). This friend declaration references the previously synthesized `CRD` pointer as the befriending class and the `D` parameter as the friend declaration. Once we have the friend declaration in place, the access checking mechanism will assess the target function like any other regular friend function.

As the last step, we register the attribute for the declaration (lines 26-28). This step is not essential, but it makes the whole procedure complete. We did this because in some future static analysis or another tool might want to process this information.

2.5.4 Related Work

Accessing private *non-static* data members via static pointers was first presented by Johannes Schaub [147]. Later it was extended by Chandra Shekhar Kumar [148] so as to use friend functions instead of static pointers. We also extended Kumar's approach making it simpler and cleaner and we based our generic macro library implementation on the simplified version. To the best of our knowledge, accessing private *static* variables had never been presented before our work.

2.5.5 Conclusion

All non-intrusive testing methods require access to the internal state of the objects under test. Our new methods are of course no exceptions. Therefore, for the sake of accessing private members, we discussed different techniques available for C++. Exploiting an exceptional language rule concerning explicit template instantiation provides an interesting way of accessing private non-static data members. We generalized the technique to access static members and member functions as well. Then we created a library to automate the access. Currently, this library is the only generic solution to access private members without violating the C++ standard.

Since our new technique above still had some shortages, we presented a more generic method to access private or protected members. Friend declarations added outside of a class could provide a full, non-intrusive solution to separate test related code from the source of the unit under test. This new language element has the

capability to work on every private asset, data, function, or type. We also realized a prototype based on C++ attributes to demonstrate the feasibility of the idea.

2.5.6 Contribution

Thesis 2 (Extending access for non-intrusive and white-box testing)**.** *I have analysed the various existing methods available for accessing private members in C++. To support non-intrusive and white-box testing I have developed two different approaches eliminating the existing drawbacks. (1) I have created a library which exploits the explicit template instantiation mechanism of C++ and this way enables access to private members. Currently, this library is the only generic solution to access private members without violating the C++ standard. (2) I have presented how friend declarations added outside of a class could provide a full, non-intrusive solution to separate test related code from the source of the unit under test. I have realized a prototype based on C++ attributes to justify the feasibility of out-of-class friends.*

thesis name	relevant publications									
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
(1) New non-intrusive testing methods	∘	∘	∘	∘					∘	
(2) **Extending access for non-intrusive and white-box testing**	•							•		•
(3) Selective friend					∘					
(4) High-level abstraction for the read-copy-update pattern						∘	∘			

Chapter 3

Selective Friend

There is a strong prejudice against the friendship access control mechanism in C++. People claim that friendship breaks the encapsulation, reflects bad design and creates too strong coupling. However, friends appear even in the most carefully designed systems, and if it is used judiciously (like using the Attorney-Client idiom) they may be a better choice than widening the public interface of the class.

In this section, we investigate how the friendship mechanism is used in C++ programs. We have made measurements on several open source projects to understand the current use of friends. Our results show various holes and errors in friend usage, like friend functions accessing only public members or not accessing members at all or the class which declare friends has no private members at all. The results also show that friend functions actually use only a low percentage of the private members they were granted to access, which is a source of errors.

These results have motivated us to propose a selective friend language construct for C++ which can restrict friendship only to well-defined members. Such a new language element may decrease the degradation of encapsulation and significantly increase the diagnostic capacity of the compiler. We have created a proof-of-concept implementation based on the LLVM/Clang compiler infrastructure to show that such constructs can be established with a minimal syntactical and compilation overhead.

3.1 Motivation

Encapsulation is one of the quintessential object-oriented programming concept (as we discussed in Chapter 1). Strictly speaking, information hiding is not a mandatory part of encapsulation [149]. Encapsulation can be achieved by using different programming conventions and good programmers have already successfully applied these practices in various languages. However, language support for

information hiding is essential in large-scale software projects, as the violation of encapsulation rules can be detected in an automated way.

In modern statically typed programming languages information hiding is usually implemented via specific access control rules. These *visibility rules* are typically compiler-checked static rules, therefore no or minimal runtime activity is involved. Most languages support module-based access control, but proposals have been made to refine the type system of these languages to implement object encapsulation too [150, 151, 152, 153, 154, 155]. In contrast, popular dynamically typed languages such as Python, Ruby, and Smalltalk provide very limited or no encapsulation at all. The implementation of object-oriented encapsulation for dynamically typed languages is an important research area [33].

In object-oriented languages, the notion of the class is the main tool for both modularization and encapsulation. Accordingly, most of the access rules define the visibility of class members. Modern programming languages export services in an anonymous way; i.e. not naming the client classes able to use those services. An important exception is the Eiffel programming language, where *selective export* possibilities provide a fine-grained description to specify which class components are visible from individual client classes [156]. Unfortunately, most current popular object-oriented languages lack this feature. Java, C#, and C++ restrict us to enrol member access into visibility categories. In C++ we can use categories, like "exporting to all clients" (`public`), "exporting to the subclasses" (`protected`), and "exporting to none" (`private`). Though Java and C# add additional categories, they still rely on anonymous category-based export.

Schärli et. al. [149] pointed out that current object-oriented programming languages are surprisingly weak in terms of encapsulation mechanism provided for programmers. Their criticism emphasizes that access rights are inseparable from classes, and are not customizable. The authors also diagnose that access rights are specified in terms of fixed client categories, typically users and heirs (i.e. using public and protected categories).

In large projects, when the number of corresponding components is growing, the class interactions form exponential accretion. Here superfluous visibility is a persistent source of design, implementation and maintenance problems. On the other hand, a fine-grained visibility strategy can utilize the full compiler possibilities of modern object-oriented languages to filter out unwanted access attempts.

The C++ programming language has a special feature called `friend` declaration to specify individual methods or full classes to access non-public members in a class. Unfortunately, a friend has permission to access every field, method, nested class, etc. with private or protected visibility. This behaviour essentially switches off all possible automated detection of access violations, i.e. when a friend accesses wrong members.

3.2 C++ Friends

In C++, *friend* is a special language element for access control mechanism [63, section 11.3]:

> A friend of a class is a function or class that is given permission to use the private and protected member names from the class. A class specifies its friends, if any, by way of friend declarations.

Example:

```
class A {
  int x = 0;
  friend void foo(A &a);
};

// Accessing private member:
void foo(A &a) { a.x = 42; }
```

A *befriending class* is a struct/class which declares one or more friend function, friend function template, friend class or friend class template. Friend declarations give special access rights to the friends, but they do not make the nominated friends members of the befriending class.

Friendship is a great tool to enhance encapsulation in some special cases [34]. We often need to split a class in half when the two halves will have different numbers of instances or different lifetimes. In these cases, the two halves usually need direct access to each other. The safest way to implement this is to make the two halves friends of each other. Friends have been used also for providing better syntactics [157]. Consider for instance the binary infix arithmetic operators: `aComplex + aComplex` should be defined logically as part of the `Complex` class. However, `operator+` has to be a free-standing function to support `aFloat + aComplex` order of arguments as well, thus we declare it as a friend. Other examples of better syntax are the stream operators (`<<`, `>>`) which often have to access the internals of a class and they have to be free functions.

Friend is an explicit mechanism for granting access, just like membership. Member functions and friend functions are equally privileged, they access every innard of a class. The major difference is that a friend function is called like `f(x)`, while a member function is called like `x.f()` [34, 158]. Stroustrup states that during the language design, a friendship declaration was seen as a mechanism similar to that of one protection domain granting a read-write capability to another. It is an explicit and specific part of a class declaration [159, 2.10].

When we design a class, we try to minimize the number of functions that access the representation of the class and try to make the set of access functions as appropriate as possible. Therefore, the first question regarding a (candidate

member) function is whether it really needs access to the class? Typically, the set of functions that need access is smaller than we are willing to believe at first [158, 19.4.2]. Meyers states that given a choice between a member function (which can access not only the private data of a class, but also private functions, enums, typedefs, etc.) and a non-member non-friend function (which can access none of these things) providing the same functionality, the choice yielding greater encapsulation is the non-member non-friend function, because it does not increase the number of functions that can access the private parts of the class [60, Item 23]. Sutter and Alexandrescu have similar statements [28, chapter 44]. In other words, if we have a friend function which does not access any private entities it weakens the encapsulation since it increases the number of functions that can access the internals. By declaring that function non-friend, the encapsulation is not weakened and the functionality can be preserved since there is no need to access private entities. Consequently, we refer a friend function (or a method of a friend class) as *erroneous* or *superfluous* if it is not used to provide better syntax — like the non-member `operator+` — and it does not access any private or protected member.

Because friends can be easily misused [159, 160] some people discourage the use of friends at all [161, 162]. The different use cases of friendship and the criticism about it have motivated us to investigate friend usage in real-world, open source projects.

3.3 Friends in Other Programming Languages

3.3.1 Java

In the Java language, there is no such thing as friendship. However, members without any access modifier (package-private) are accessible via all those classes which are declared in the same package [163]. This mechanism cannot control access in such a fine-grained way as friendship, but for most of the cases, it is good enough. For example, a white-box test [143, 144] might want to have access to the class it is testing. This can be achieved by declaring the members of the class package-private and by adding the test to the same package.

For a more sophisticated, friendship like access control we can use the workaround presented at Figure 3.1 [164]. Here, class `Owner` has a privileged function which shall be accessed only by the `User` class. The nested and public `User.Key` class has a private constructor which is accessible only in the parent class (`User`), therefore the `User.Key` object can be constructed only by the `User` class. The `User.usePrivileged()` function uses a static instance of the mentioned Key. The `Owner.privileged()` function requires an instance of the `User.Key` class to be

```
import java.util.*;
import java.lang.*;
import java.io.*;

class Owner {
  public static void privileged(User.Key l) {
    l.hashCode();
    System.out.println("Privileged␣function␣called.");
  }
}

class User {
  public static class Key {
    private Key() {}
  }
  private static Key key = new Key();
  public static void usePrivileged() {
    Owner.privileged(key);
  }
}

class Main {
  public static void main(String[] args) throws java.lang.Exception {
    User.usePrivileged();
    // Owner.privileged(null); // runtime error
    // Owner.privileged(new User.Key());
    //                  ^^^^^^^^^^^^compile error
  }
}
```

Figure 3.1: Friendship like access control in Java

passed. Putting this together, only the methods of the `User` class will be able to call `Owner.privileged()`. If we call `Owner.privileged()` with a null pointer, then we will receive a runtime error. If we try to create an instance of `User.Key` not in the scope of `User` class, then we will get a compile error.

This technique cannot be implemented either in C++ or in C#, because it relies on a special access rule, which is unique for Java. Namely, an enclosing class can access its nested classes' private members [165, 166]. With this approach, we can precisely control which classes can access privileged functions of a class. By evolving further this technique, we can also control which specific member functions are accessible and this is more fine-grained access control than C++ friendship is. We can restrict access of friends only to a specific set of member functions in C++ as well, with a similar idea which also uses private constructors and special `Key` classes. Albeit, those `Key` classes need to declare some auxiliary friends. Later in section 3.6.3 we elaborate this approach. This methodology is capable of handling only member function access, but it cannot handle member field access.

3.3.2 CSharp

In C# there is no friend keyword as well, though we can define members as `internal`, thus making them available to the current assembly [167]. This is very similar to what Java has with the package-private access level.

Also, there is the `InternalsVisibleToAttribute` with which we can achieve a behaviour that is very similar to friendship [168]. With this attribute, we are able to declare other assemblies to be friends of our assembly. The access control is happening on assembly level and we cannot be more specific to control the access class by class [169].

Figure 3.2 is an example that uses the `InternalsVisibleToAttribute` attribute to make an internal member of an unsigned assembly visible to another unsigned assembly. The attribute ensures that the inner `IsFirstLetterUpperCase` method of the `StringLib` class in an assembly named `UtilityLib` is visible to the code in an assembly named `Friend2`. We can see an example in Figure 3.3 which provides the source code for the `Friend2` assembly. Note that if we are compiling in C# from the command line, we must use the /out compiler switch to ensure that the name of the friend assembly is available when the compiler binds to external references.

3.3.3 Other Languages

D is a systems programming language with C-like syntax and static typing. It combines efficiency, control and modelling power with safety and programmer pro-

```
using System;
using System.Runtime.CompilerServices;

[assembly: InternalsVisibleToAttribute("Friend2")]

namespace Utilities.StringUtilities
{
   public class StringLib
   {
      internal static bool
      IsFirstLetterUpperCase(String s)
      {
         string first = s.Substring(0, 1);
         return first == first.ToUpper();
      }
   }
}
```

Figure 3.2: Use of InternalsVisibleToAttribute in CSharp (UtilityLib.dll)

```
using System;
using Utilities.StringUtilities;

public class Example
{
   public static void Main()
   {
      String s = "The␣Sign␣of␣the␣Four";
      Console.WriteLine(StringLib.IsFirstLetterUpperCase(s));
   }
}
```

Figure 3.3: Use of InternalsVisibleToAttribute in CSharp (Friend2 assembly)

```
module X;
class A {
  private:
    static int a;

  public:
    int foo(B j) { return j.b; }
}
class B {
  private:
    static int b;

  public:
    int bar(A j) { return j.a; }
}
int abc(A p) { return p.a; }
```

Figure 3.4: Accessing private members in D (within the same module)

ductivity. In this language, friend access is implicit in being a member of the same module [170]. This means a member which is declared private can be accessed by other classes of the same module (Figure 3.4). The private attribute prevents only other modules from accessing the members.

Rust is another systems programming language [171, 172]. In addition to public and private, Rust allows users to declare an item as visible within a given scope (Figure 3.5). This mechanism is similar to friendship, but we grant access to whole modules not to just simple functions or classes. The rules are as follows:

- `pub(in path)` makes an item visible within the provided path. `path` must be a parent module of the item whose visibility is being declared.
- `pub(crate)` makes an item visible within the current crate. (A crate is a unit of compilation and linking, as well as versioning, distribution and runtime loading.)
- `pub(super)` makes an item visible to the parent module.
- `pub(self)` makes an item visible to the current module.

Script languages like Python and Perl do not have any access restriction mechanism [173, 174]. However, there is a convention to name class internal variables with an underscore prefix. The lack of access restriction sometimes results in unbeneficial workarounds. For instance, in case of one Python implementation of the Singleton pattern an exception is raised if the Singleton object is already instantiated (since there is no way to make the constructor private) [175].

```
pub mod outer_mod {
    pub mod inner_mod {
        // This function is visible within ‘outer_mod‘
        pub(in outer_mod) fn outer_mod_visible_fn() {}

        // This function is visible to the entire crate
        pub(crate) fn crate_visible_fn() {}

        // This function is visible within ‘outer_mod‘
        pub(super) fn super_mod_visible_fn() {
            // This function is visible since we’re in the same ‘mod‘
            inner_mod_visible_fn();
        }

        // This function is visible
        pub(self) fn inner_mod_visible_fn() {}
    }
    pub fn foo() {
        inner_mod::outer_mod_visible_fn();
        inner_mod::crate_visible_fn();
        inner_mod::super_mod_visible_fn();

        // This function is no longer visible
        // since we’re outside of ‘inner_mod‘:
        //  Error! ‘inner_mod_visible_fn‘ is private
        //inner_mod::inner_mod_visible_fn();
    }
}
```

Figure 3.5: Rust: declaring items visible within a given scope

3.4 Measurement

For a better understanding of the current use of friends in C++, we decided to carry out empirical research. We were interested in (i) how many private members are accessed by friend functions, (ii) what is the ratio of accessed per non-accessed private members, (iii) are there indications of erroneous friend usage. To execute the measurement we implemented a utility based on the LLVM/Clang[1] compiler infrastructure' LibTooling and LibASTMatchers libraries [176]. With the help of these libraries, we can derive statistics from the abstract syntax tree (AST) of the examined source files. Our utility is publicly available [19].

3.4.1 Description of the Measurement Algorithm

We can collect the statistics just for one translation unit (TU), or for a whole project with several translation units. If there are more than one TUs in the measure, we pay attention to not count twice those friend declarations whose definitions are in headers.

During the measurement, we avoid collecting information on unused (i.e. neither instantiated nor explicitly specialized) templates in all contexts. If the befriending class is a class template, then it must have at least one instantiation/specialization to participate in the measure. We do not measure partial specializations.

We generate the statistics from measurement entries, which are stored for each friend entity. We store a *function measurement entry* for each friend function definition. If that friend declaration is a function template, then we create an entry for each different template specialization/instantiation. It means we do not examine the primary template itself, but only the specializations/instantiations. So, not used (primary) function templates will not affect the collected statistics.

In the case of friend classes, we store a *class measurement entry*. If we find a friend class template, then we examine only its specializations/instantiations. For every friend class or class template specialization/instantiation, we examine all of its member functions and member function templates and we create a function measurement entry for each of them. The member function templates are handled similarly as we handle the simple friend function templates, i.e. only the specializations/instantiations are checked, not the primary function template.

If the friend class or class template specialization/instantiation has nested classes, then we collect statistics from all of the functions and function templates of the nested classes. The rules regarding the function instantiation and specialization are the same as in case of non-nested classes. If the nested class happens

[1]The LLVM compiler infrastructure is briefly presented in Appendix B.

```
1  class A {
2    template <typename T>
3    friend void func(T, A& a);
4  };
5  template <typename T>
6  void func(T, A& a) {}
7
8  // Full(explicit) specialization
9  template <> void func<double>(double, A& a) {}
10
11 // Explicit instantiations
12 template void func<int>(int, A&);
13 template void func<double>(double, A&);
14
15 // Implicit instantiation
16 void foo() { A a; func<double>(1.0, a); }
17
18 // An other primary template
19 template <typename T> void func(T*, A&) {}
```

Figure 3.6: A more complex example for measurement

to be a class template, then we check only the specializations/instantiations of it. Again, in cases of class templates, we do not check partial specializations. Only explicit specializations and explicit/implicit instantiations are counted.

For example, consider the below code:

```
class A {
  friend class B;
};
class B {
  void func(A &a) {}
  template <typename T>
  class C {
    void func(A &a) {}
  };
};
template class B::C<int>;
```

Here, the measurement will contain two class measurement entries; one for `class B` and one for `class B::C<int>` instantiation. Each class measurement entry contains one function measurement entry.

A more complex example follows in Figure 3.6. Examining this translation unit, we will have two function measurement entries. One for the specialization/instantiation with `double` and one for the instantiation with `int`. The instantiations in line 13 and 16 are superfluous instantiations since the compiler has already created the corresponding representation for those instantiations via the specialization in line 9. The last line of the example (line 19) defines a new primary template which is independent of the friend function template.

Note, there is no such thing as partial function template specialization in C++, therefore

```
template <typename T> void func(T, A& a) {}
template <typename T> void func<T*>(T*, A&);
```

would be illegal. What is more, it is advised to not specialize function templates at all [28, chapter 66].

Each function measurement entry contains the number of used private or protected

- member variables (including static variables),
- member functions (including static functions),
- types

in that particular function or function template specialization/instantiation. We refer private or protected member variables, member functions and nested types as *private entities*. Also, each entry contains the number of private or protected entities in the corresponding befriending class.

For instance:

```
class A {
  int a = 0;
  int b;
  int c;

  friend bool operator==(A x, A y)
  {
    return x.a == y.a;
  }
};
```

Here, we have one function measurement entry for `operator==`; the number of used private member variables is 1 (variable `a`); the number of private member variables in the befriending class is 3 (variables `a`, `b`, `c`).

All the statistics are derived from function measurement entries. Statistical values are calculated separately for friend classes and for friend functions.
Under the term *friend function instances*, we refer:

- friend functions
- friend function template specializations/instantiations

of the befriending class. We name those friend function instances which use at least 1 private entity as *correct friend function instances*. Under the term *friend class function instances* we refer:

- friend classes' member functions
- friend classes' nested classes' member functions
- the specializations/instantiations of the above two, if they happen to be function templates
- member functions of friend class template specializations/instantiations
- the specializations/instantiations of the above, if it happens to be a function template
- member functions of nested class template specializations/instantiations of friend class template specializations/instantiations
- the specializations/instantiations of the above, if it happens to be a function template

of the befriending class.

Under the term *friendly function instances* we refer all the friend function instances plus all the friend class function instances of the befriending class. We name those friendly function instances which use at least 1 private entity as *correct friendly function instances*. The *private usage* of a friendly function instance is the number of the used private entities in that particular function. The *private usage ratio* of a correct friendly function instance is the number of the used private entities divided by the number of all the private entities of the befriending class. We interpret this number only on correct friendly functions, so it is always greater than 0.0 and less than or equal to 1.0 (as a correct friendly function uses at least 1 private entity, which implies that its befriending class has at least 1 private entity). The private usage of the `operator==` in the previous example is 1; the private usage ratio is $\frac{1}{3}$.

Avoiding duplicated measurement entries

We must not count friend declarations multiple times if they are defined in a header and there are multiple source files including this header. Consider the following 3 files:

```
// header file: a.h
class A { friend void func(); }; void func(){};

// source file: TU_A.cpp
#include "a.h"

// source file: TU_B.cpp
#include "a.h"
```

In this case, we provide only one function measurement entry. That entry is reachable via an associative container and identified by the source location of the friend declaration (`a.h:1:23`).

We need to provide extra care in order to avoid the counting of the same instantiations or specializations multiple times. We need to be attentive both when the befriending class is a template and when the friend function itself is a template. For example, consider the following header file:

```
1  // header file: a.h
2
3  template <typename T> class A;
4
5  template <typename T> void func(A<T> &a);
6
7  template <typename T> class A {
8    int a = 0;
9    int b;
10   int c;
11
12   // refers to a full specialization
13   // for this particular T
14   friend void func<T>(A &a);
15 };
16
17 template <typename T>
18 void func(A<T>& a) {
19   a.a = 1;
20 }
```

We forward declare the class template `A` (line 3), then we forward declare the function template `func` that takes a reference to the object of one instantiation of the class template `A` (line 5). Later, we define the class template (line 7-15). Inside the class definition, we declare a friend function for each instantiation of `A` (line 14). The definition of the function template follows (line 17-20).

Then, let us imagine we have the following two source files, each of them is compiled into a separate translation unit:

```
// source file: TU_A.cpp
#include "a.h"
template void func(A<int>& a);

// source file: TU_B.cpp
#include "a.h"
template void func(A<int>& a);
```

Both of them contain the very same implicit instantiation of the class template, but our measurement shall not count twice. Also, the function template is explicitly instantiated twice, but we shall avoid adding the very same function measurement entry twice to the friend declaration. To achieve this, we need to identify the instantiations in a unique way. We use the diagnostic name of the identifiers,

which is the human readable form of the mangled names. We get this by calling the `getNameForDiagnostic()` function on the different `Decl` classes provided by Clang. The above set of the three files compiled into a project produces the following function measurement entry:

```
befriending class: A<int>
friendly function: func<int>
friendDeclLoc: a.h:14:15
defLoc: a.h:18:6
diagName: func<int>
usedPrivateVarsCount: 1
parentPrivateVarsCount: 3
usedPrivateMethodsCount: 0
parentPrivateMethodsCount: 0
types.usedPrivateCount: 0
types.parentPrivateCount: 0
```

So we have one function measurement entry registered for the one friend declaration (which is identified by its source location).

However, each friend function template could have different specializations with their own definition. Therefore, for one friend declaration, we should register more than one measurement entries. So we store measurement entries for such different specializations in an associative container, with a tuple of the name of the befriending class and the actual specialization as a key. (We need to be able to search this container to avoid duplicates.) For instance, if the header file is the same as before but the two source files are the following then we will have two function measurement entries:

```
// source file: TU_A.cpp
#include "a.h"
template void func(A<int>& a);

// source file: TU_B.cpp
#include "a.h"
template <class T>
struct Z { using type = char; };
template <class T>
using XYZ = Z<T>;
template void func(A<XYZ<char>::type>& a);
```

We can see that the canonical type of `XYZ<char>::type` is actually simple `char`. Therefore, we get one entry for `func<char>` and one for `func<int>`:

```
befriending class: A<char>
friendly function: func<char>
friendDeclLoc: a.h:14:15
defLoc: a.h:18:6
diagName: func<char>
// ...

befriending class: A<int>
friendly function: func<int>
friendDeclLoc: a.h:14:15
defLoc: a.h:18:6
diagName: func<int>
// ...
```

The keys with which we store these entries are the pairs (`A<char>`, `func<char>`) and (`A<int>`, `func<int>`). The layout of the measurement structure for friend functions has the following form:

```
using FriendDeclId = std::string;
using BefriendingClassInstantiationId = std::string;
using FunctionTemplateInstantiationId = std::string;
using FuncResultKey = std::pair<BefriendingClassInstantiationId,
                                FunctionTemplateInstantiationId>;
using FuncResultsForFriendDecl = std::map<FuncResultKey, FuncResult>;

std::map<FriendDeclId, FuncResultsForFriendDecl> FuncResults;
```

The above consequences are similar in case of friend classes, therefore we just present the layout of the measurement structure for the friend classes:

```
struct ClassResult {
  std::string diagName;
  FuncResultsForFriendDecl memberFuncResults;
};
using ClassTemplateInstantiationId = std::string;
using ClassResultKey = std::pair<BefriendingClassInstantiationId,
                                 ClassTemplateInstantiationId>;
using ClassResultsForFriendDecl =
    std::map<ClassResultKey, ClassResult>;

std::map<FriendDeclId, ClassResultsForFriendDecl> ClassResults;
```

Special purpose friend functions

Some degenerated use of friendship has evolved over the years. One such usage of friend functions is to define free functions inside the declaration context of the befriending class, i.e. to define free functions in-class. In this case, access control is not considered at all, only the secondary property of a friend function definition is used. For example the Boost.Operators [177] library uses friend functions and CRTP [178] to generate free functions for client classes which derive from the library classes.

In the following listing, a greater-than operator is automatically added, even though there is no declaration because the greater-than operator can be implemented using the already defined less-than operator [179]:

```
struct animal : public boost::less_than_comparable<animal> {
  std::string name;
  int legs;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
  bool operator<(const animal &a) const { return legs < a.legs; }
};

int main() {
  animal a1{"cat", 4};
  animal a2{"spider", 8};
  std::cout << std::boolalpha << (a2 > a1) << '\n';
}
```

The corresponding generator class has the following form:

```
template <class T, class B = ::boost::detail::empty_base<T>>
struct less_than_comparable1 : B {
  friend bool operator>(const T &x, const T &y) { return y < x; }
  // ...
};
```

We might wonder why the library does not provide a free function template instead of an in-class defined friend function, like below:

```
template <class T>
bool operator>(const T &x, const T &y) {
  return y < x;
}
```

If it did that, then there would be a possible instantiation for all types, not just for the type we really want to have the greater-than operator generated.

Another special purpose use of friends is to overcome some language constraints during template argument deduction. For example, let us assume we want to support implicit conversions on all operands of an operator of a class template [60]:

```
template <typename T> class Rational {
public:
  Rational(const T &numerator = 0, const T &denominator = 1);

  const T numerator() const;
  const T denominator() const;
  // ...
};

template <typename T>
const Rational<T> operator*(const Rational<T> &lhs,
                            const Rational<T> &rhs) {
  // ...
}
```

We want the code below to compile:

```
Rational<int> oneHalf(1, 2);
Rational<int> result = oneHalf * 2; // error!
```

However, the second line will not compile. The reason behind that is the compiler tries to deduce the `T` template type parameter of `operator*`, but it cannot. The deduction succeeds for the first parameter (`oneHalf`), but it fails for the second parameter. We might expect the compiler to use `Rational<int>`'s non-explicit constructor to convert `2` into a `Rational<int>`. But it does not do that, because implicit type conversion functions are never considered during template argument deduction.

The solution for this problem is to define the `operator*` as a friend function inside the body of the `Rational` class template:

```
template <typename T> class Rational {
public:
  // ...

  friend const Rational operator*(const Rational &lhs,
                                  const Rational &rhs) {
    // ...
  }
};
```

Now the call to `operator*` will compile, because when the object `oneHalf` is declared to be of type `Rational<int>`, the template class `Rational<int>` is instantiated, and as part of that process, the friend function `operator*` that takes `Rational<int>` parameters is automatically declared. As a declared function (not a function template), compilers can use implicit conversion functions (such as `Rational`'s non-explicit constructor) when calling it, and that is how they make the call succeed. As Meyers writes in [60]:

> The use of friendship has nothing to do with a need to access non-public parts of the class. In order to make type conversions possible

> on all arguments, we need a non-member function; and in order to have the proper function automatically instantiated, we need to declare the function inside the class. The only way to declare a non-member function inside a class is to make it a friend.

This use relies on the side effects of friend declarations, and it falls far away from the original intention of friendship, which was to provide sophisticated access control.

We refer a friend function as *Meyers candidate* if it has the following properties:

- It has zero private usage.
- Its befriending class is a class template.
- The function is defined inside (in-class) the befriending class.

The friend functions in both of the above examples (`less_than_comparable1 ::operator>` and `Rational::operator*`) fulfills these properties. Meyers candidates are very likely to be using the `friend` declaration specifier to define in-class free functions for a specific purpose and not for accessing private entities.

3.4.2 Measurement Results

We have measured four open source projects:

1. Boost Libraries [180], version 1.56.0, ∼2.0 million lines of C/C++ code
2. LLVM and Clang [82], version 3eec7e6 (llvm.org/git/clang.git), ∼2.3 million lines of C/C++ code
3. ITK [181], version 4d37786 (itk.org/ITK.git), ∼1.0 million lines of C/C++ code
4. Qt (qtbase package, core module) [182], version 5.6.3, ∼200 thousand lines of C/C++ code

The detailed measurement logs can be accessed publicly [19].

Figure 3.7 shows the distribution of private usage in friend function instances for these projects. We can see for example in case of the Boost libraries there are around 95 friend functions which use one private entity. With the shaded (white) bar we display the number of the Meyers candidates. We do not draw bars for those results where the number of friends is zero; for instance, there is no bar for 7 in case of the Clang project. We can see that there are lots of friend function instances which do not use any of the private entities (zero private entity usage). There can be two reasons behind this:

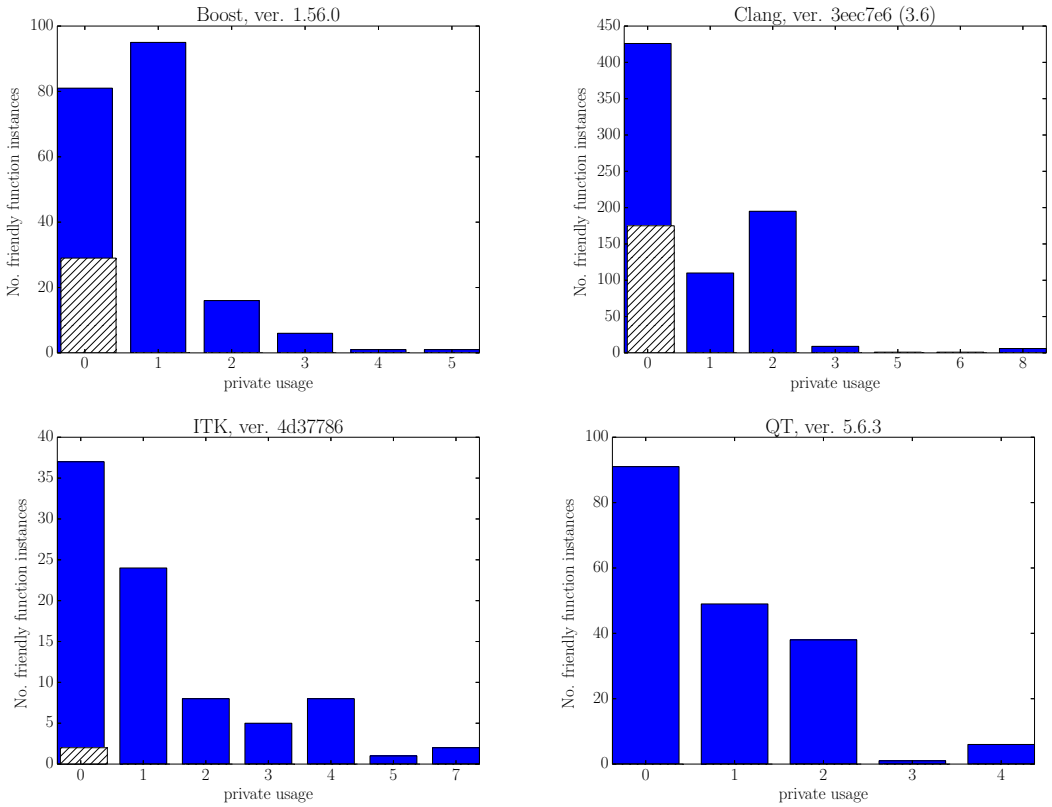


Figure 3.7: Private usage in functions

1. The befriending class does not have any private entities.

2. The friend function instance simply does not use any of the private entities. Note, those friend function instances which are in this category might be in the previous category as well because a friend function cannot use any of the private entities if the befriending class does not have any.

A surprisingly big portion of the friend function instances are not using any of the private or protected members. They are declared as friends either by mistake or during maintenance their implementation has changed to not access any privates. Or they have been declared as friends for some special reason which is not connected to access control, i.e. they are Meyers candidates.

According to Stroustrup we should strive to minimize the number of functions that access internals [158, 19.4.2]. Meyers states the encapsulation is greater if there are fewer functions that can access the private parts of the class [60, Item 23]. Consequently, we consider all those non-Meyers-candidate friend functions as erroneous friends which have zero private usage.

Those friend functions who are Meyers candidates shall not be counted as erroneous friends. Even though they do not access any private entities they might have been declared as friends to define a free function. However, we cannot know if this is the only true case. It might as well be possible, that the function had accessed privates previously, but during maintenance, it has been changed to not access privates anymore. Nevertheless, the number of Meyers candidates gives us the upper limit of the error of our measurement regarding erroneous friend functions.

We conclude that even considering the Meyers candidates there can be many friend functions or classes which are declared as friends superfluously, this way weakening the encapsulation. Therefore, we created a tool [19] with which we can list all the possibly erroneous friend declarations in a project. More specifically this tool can list

1. All those befriending classes which have at least one friend declaration but itself does not have any private entities. Note we do not list a class if all of its friend declarations are functions and all of those functions are Meyers candidates.

2. All those friend classes whose all member functions do not access any private entities in the befriending class (and the befriending class has private entities).

3. All the friend functions which might not be friend because they do not access any private entities. We list only those functions whose befriending class have private entities and it is not a Meyers candidate.

```
struct uchar_wrapper {
  //... // there is no any access specifier

  // Friend function definition
  friend std::ptrdiff_t
  operator-(uchar_wrapper const &lhs, uchar_wrapper const &rhs) {
    return lhs.base_cursor - rhs.base_cursor;
  }

  //...
};
```

Figure 3.8: An erroneous friend declaration in Boost

(Note, this tool is also based on Clang LibTooling library and shares a lot of common source code with the statistical measurement tool). With the help of the tool we could identify those friends in the Boost, Clang, ITK and Qt projects that were declared as friend by mistake. For instance, in the ITK library, there is a befriending class, which declared the `operator<<` as a friend function:

```
class GDCM_EXPORT Sorter {
  friend std::ostream &
  operator<<(std::ostream &_os, const Sorter &s);

public:
  // ...
  void Print(std::ostream &os) const;

protected:
  std::vector<std::string> Filenames;
  // ...
};

inline std::ostream &operator<<(std::ostream & os, const Sorter &s) {
  s.Print(os);
  return os;
}
```

Still, the definition of `operator<<` uses only the public `Print()` function and does not use any of the private entities.

We can find another example for erroneous friend usage in the Boost libraries as well, listed in Figure 3.8. The `uchar_wrapper` struct does not have any explicit access specifier in the body of the class definition, i.e. all the members are public. Still, the `operator-()` is declared as friend. These cases safely can be considered as misuses of the friend construction. Note, currently there is no language construction in C++ to enable the compilers to detect and warn about such situations.

Figure 3.9 shows the distribution of the private usage in friend class function instances for Boost, Clang, ITK and in Qt corelib. The number of friend class

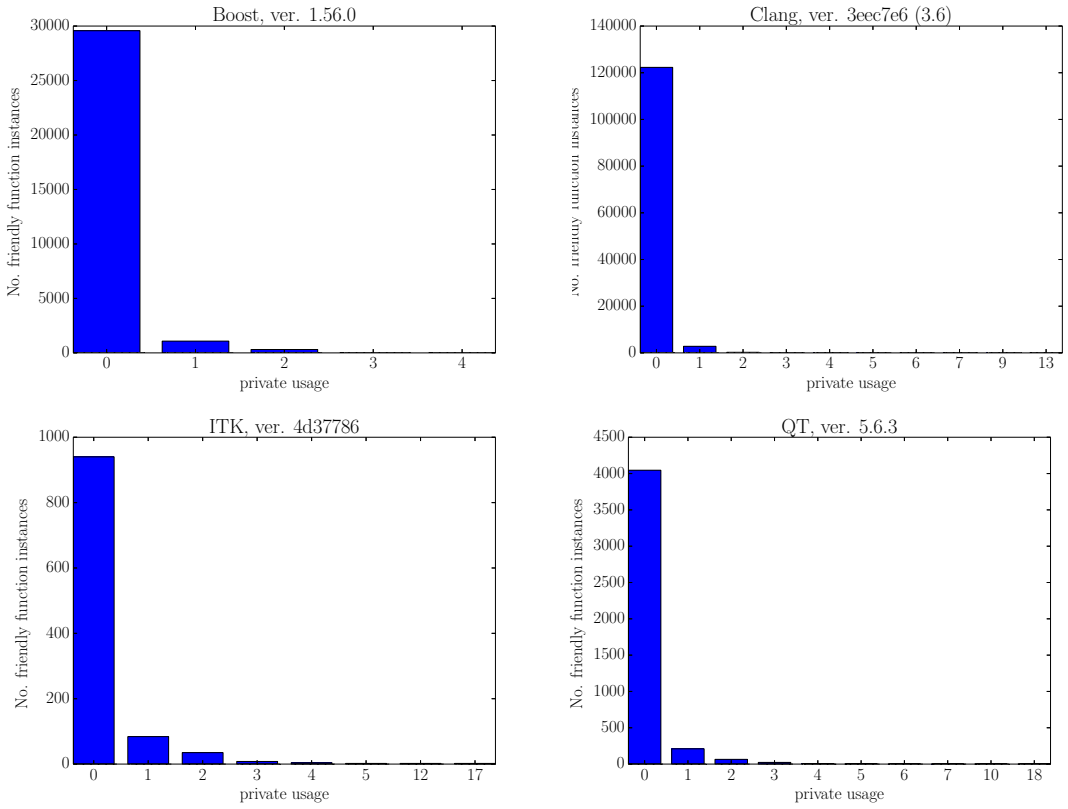


Figure 3.9: Private usage in classes

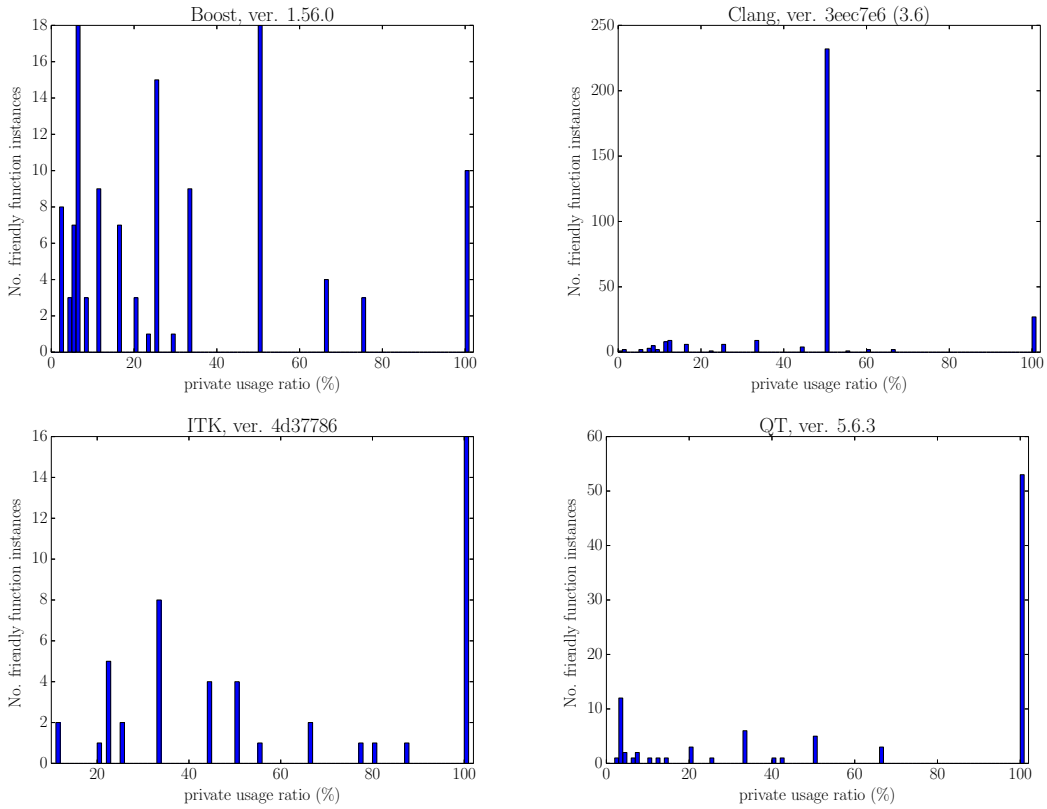


Figure 3.10: Private usage ratio in functions

function instances with zero private entity usage is quite high. In all four libraries, the number of these class function instances is significantly larger than the number of the correct class function instances. There is a huge number of member functions in friend classes which does not use any private members but they have access for all of the private entities of the befriending class. Compared to friend functions, the ratio of non-correct / correct function instances is way higher in the case of friend classes. This means that friend classes provide access for a wide range of functions that are potentially unrelated to the befriending class. Therefore we consider the use of friend classes more harmful to encapsulation than the use of friend functions.

Figure 3.10 presents the average private usage ratio of correct friend function instances while figure 3.11 displays the average private usage ratio of correct friend class function instances. We can see, there are lots of the correct friendly function instances which are just accessing only a small portion of the befriending class' private entities. Also, in Figures 3.7 and 3.9 we can see that the vast majority of the correct friendly function instances access only 1-4 private entities. I.e. those

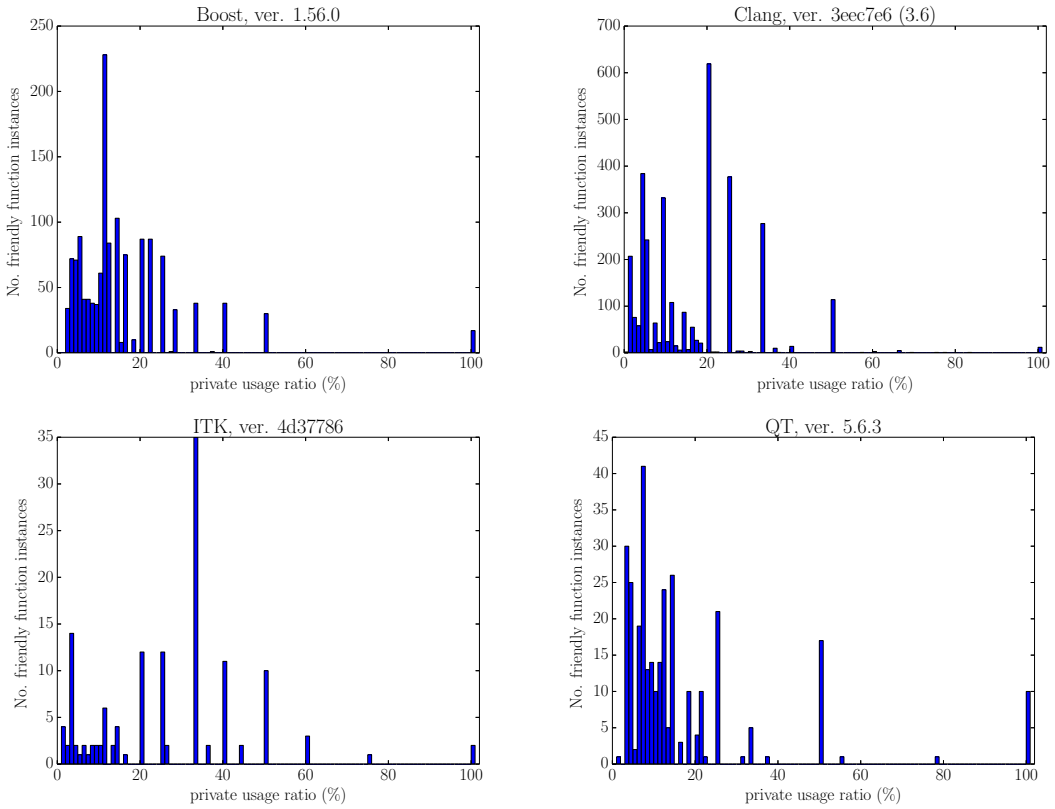


Figure 3.11: Private usage ratio in classes

functions which do access internals usually do not access more than a few members. In other words, the library authors allowed unnecessary access to a large number of private entities.

3.5 Selective Friend

3.5.1 A New Lingual Element

Meyers states the encapsulation is greater if the number of functions that can access the private parts of the class is fewer [60, Item 23]. Similarly, we state that the encapsulation is greater if the number of accessible private entities is fewer because in that case, the accessible private part of the class is smaller.

We have seen in section 3.4.2 that in Boost, Clang, ITK and Qt the vast majority of correct friendly function instances access only 1-4 private entities although the number of private entities in the befriending class is usually much higher. Hence, we conclude there are many friend functions who access only a small subset of all the private members and this way the encapsulation is weakened. To overcome this issue, we propose a new language construct which enables friends to access only a restricted set of explicitly named private entities. Let us consider the syntax:

```
class A {
  int x = 0;
  int y = 0; // expected-note
             // {{implicitly declared private here}}
  friend for (x) void func(A &a);
};

void func(A &a) {
  a.x = 1; // OK
  a.y = 1; // expected-error
           // {{'y' is a private member of 'A'}}
}
```

During the compilation, we would expect an error at the line when we attempt to access `a.y` private member because we explicitly stated that `func` can access only `A::x`.

An alternative syntax conforming to the current C++17 standard is using attributes:

```
[[friend_for(x)]] friend void func(A &a);

// Access of multiple members
[[friend_for(x, y)]] friend void func(A &a);
```

Even without any change to the current C++ standard, the attribute-based `friend_for` implementation could be useful. The tools (compilers and external

tools) that support it could provide proper diagnostics when the semantics of the attribute is violated. The tools which do not support it would just simply ignore the `friend_for` specification. C++17 compliant compilers are required to silently ignore unknown attributes [37, 10.6.1/6], whereas C++14 compliant compilers may produce warnings [63, 7.6.1/5]. A proof-of-concept implementation using attributes is publicly available at [20].

Description of the Implementation

The realization of the attribute [20] is based on the LLVM/Clang[2] compiler infrastructure (version 3.6.0) [82]. The goal of this implementation is to prove that the idea is feasible, not to provide a full-featured perfect realization. Therefore, we add some restrictions about the functionality and we do not implement proper error handling. We allow only functions to be declared as selective friends. We can select only one member to be the target of a selective friend. The argument of the attribute has to be a unary expression which can be parsed as a pointer to a member. Also, we use the GCC `__attribute__` syntax because the standard `[[attribute]]` syntax implementation is not complete in the used Clang version. Thus, the prototype handles the following syntax:

```
__attribute__((friend_for(&A::x))) friend void func(A &a);
```

We define our new attribute in Clang's `Attr.td` file:

```
def SelectiveFriend : InheritableAttr {
  let Spellings = [GCC<"friend_for">];
  let Args = [ExprArgument<"Expr">];
  let Subjects = SubjectList<[Function]>;
  let Documentation = [Undocumented]; }
```

`Spellings` defines the list of the supported attribute syntaxes, this time it is only the GCC style. The attribute syntax also defines the name of the attribute, in our case, it is `"friend_for"`. `Args` specifies the list of the attribute arguments. Our attribute has only one argument which is an expression. This argument refers to a unary expression which takes the address of a member. `Subobjects` describe the list of the lingual elements that might have this attribute. In this case, we allow only functions to have it. Note, there is no way to specify an attribute to be attachable only to friend function declarations, thus we had to be satisfied with the `Function` declarations as subjects.

Once we have the attribution definition in place then the Clang infrastructure will generate all the necessary parsing code. What left is to define the semantic action for the new attribute and to hook that action into the existent compiler machinery. As for the hooking, we need to add a new function call in the `ProcessDeclAttribute` function:

[2]The LLVM compiler infrastructure is briefly presented in Appendix B.

```
static void ProcessDeclAttribute(Sema &S, Scope *scope, Decl *D,
                                 const AttributeList &Attr,
                                 bool IncludeCXX11Attributes) {
  // ...
  case AttributeList::AT_SelectiveFriend:
    handleSelectiveFriendAttr(S, D, Attr);
    break;
  // ...
}
```

This function is dedicated to apply a specific attribute to the specified declaration if the attribute applies to declarations. (Our attribute applies to function declarations.)

The semantic action for the new attribute is defined as follows:

```
1  static void handleSelectiveFriendAttr(
2      Sema & S, Decl * D,
3      const AttributeList &Attr) {
4
5    // TODO Add error handling, when D is not a
6    // FriendDecl
7
8    Expr *E = Attr.getArgAsExpr(0);
9
10   D->addAttr(
11       ::new (S.Context) SelectiveFriendAttr(
12           Attr.getRange(), S.Context, E,
13           Attr.getAttributeSpellingListIndex()));
14 }
```

The parameter `S` holds a reference of the monumental `Sema` class which is responsible for semantic analysis and AST building in the Clang compiler. The parameter `D` represents the declaration which has the attribute. The attribute itself is described with the `Attr` parameter. First, we retrieve the expression which is connected to the argument of the attribute (line 8). Note, we skip the handling of the erroneous case when the user attaches this attribute to a non-friend function. The second step is to create a `SelectiveFriendAttr` node in the AST (lines 11-13). Then we register the new node to the function declaration with `addAttr` (line 10).

Clang has a semantic action associated for some of the production rules of the `postfix-expression` non-terminal in the C++ grammar [63, appendix 4]. These production rules are describing the syntax of member access:

```
postfix-expression:
  postfix-expression . [template] id-expression
  postfix-expression -> [template] id-expression
```

In the associated semantic action, Clang checks whether the member access expression has privileges to access the actual member. In the process of checking the privileges, Clang eventually calls the `GetFriendKind()` function. This function iterates over all the friend declarations of the class that the member access expression refers to:

```
1  static AccessResult GetFriendKind(
2      Sema & S, const EffectiveContext &EC,
3      const AccessTarget &Target,
4      const CXXRecordDecl *Class) {
5
6    AccessResult OnFailure = AR_inaccessible;
7
8    // Okay, check friends.
9    for (auto *Friend : Class->friends()) {
10     switch (MatchesFriend(S, EC, Friend)) {
11     case AR_accessible:
12       return AR_accessible;
13
14     case AR_inaccessible:
15       continue;
16
17     case AR_dependent:
18       OnFailure = AR_dependent;
19       break;
20     }
21   }
22
23   // That's it, give up.
24   return OnFailure;
25 }
```

The `S` parameter is a reference to the global `Sema` class. `EC` refers to the list of the enclosing functions and/or classes (up to the highest file scope) in which the member access expression is located. The `Target` parameter represents the member that the member access expression refers to. `Class` refers to the class that the member access expression deals with. This class is not necessarily a befriending class, it might not have any friend declarations at all. For each iteration (line 10), Clang checks for the iterated friend declaration whether any of the enclosing functions or classes (`EffectiveContext`) of the actually parsed member access expression is equal to that friend declaration:

```
static AccessResult MatchesFriend(Sema &S, const EffectiveContext &EC,
                                  FunctionDecl *Friend) {
  // ...
  for (SmallVectorImpl<FunctionDecl *>::const_iterator
           I = EC.Functions.begin(),
           E = EC.Functions.end();
       I != E; ++I) {
    if (Friend == *I)
      return AR_accessible;
    // ...
  }
  // ...
}
```

If that equality holds, that means the actual member access expression is valid, because one of the parent enclosing scopes is either a friend function or a friend class.

The point which we chose to intervene into Clang's original access checking mechanism is in the `GetFriendKind()` function:

```
1   // Okay, check friends.
2   for (auto *Friend : Class->friends()) {
3     switch (MatchesFriend(S, EC, Friend)) {
4     case AR_accessible:
5       switch (SelectiveFriendConstraint(Friend,
6                                         Target)) {
7       case AR_accessible:
8         return AR_accessible;
9       case AR_inaccessible:
10        continue;
11      default:
12        assert(false &&
13               "should␣not␣reach␣this␣point");
14      }
15
16    case AR_inaccessible:
17      // ...
18    }
19  }
```

Whenever the `MatchesFriend()` function reports that the actual member is accessible then we review its decision by checking if there is a selective friend attribute (line 5-14).

Figure 3.12 shows how the constraint on the selective friend attribute is implemented. The `Friend` parameter refers to the friend declaration. The `Target` parameter represents the member that the actual member access expression refers to. First (line 5-9), we check whether we can get a `NamedDecl` pointer from the friend declaration. If we cannot get such a pointer that means the friend declaration refers to a class, not a function. Since we do not handle classes we return with `AR_accessible`, i.e. we do not do any restriction. Then we cast the `NamedDecl` into a `FunctionDecl`. If the cast is not successful that means either we deal with a function template or with a class template. If it turns out the friend declaration refers to a class template then again we leave intact the original access checking mechanism. Otherwise, if it is a function template then we get the pointer to the underlying `FunctionDecl` (line 11-22). Finally (line 24-35), we retrieve the selective friend attribute from the function if it has any. From the expression that is wrapped into the attribute, we get the AST node that holds the information about the unary operator expression. From the unary operator, we get the AST node (`DeclRefExpr`) that refers to the "address of" operator (`&`). Note, the used cast operations will abort the program execution if the cast cannot succeed; this abort will not happen until we do casts that are consistent with the attribution definition and its semantic action. Then we check whether this referenced declaration (`DRef`) is equal to the declaration of the member that the member access expression refers to (`Target`). If they are not equal that means the actually investigated member

```
1  static AccessResult SelectiveFriendConstraint(
2      FriendDecl * Friend,
3      const AccessTarget &Target) {
4
5    NamedDecl *ND = Friend->getFriendDecl();
6    // handling of friend classes not implemented
7    if (!ND) {
8      return AR_accessible;
9    }
10
11   FunctionDecl *FD = dyn_cast<FunctionDecl>(ND);
12   if (!FD) {
13     FunctionTemplateDecl *FTD =
14         dyn_cast<FunctionTemplateDecl>(ND);
15     // handling of friend class templates not
16     // implemented
17     if (!FTD) {
18       return AR_accessible;
19     }
20     FD = FTD->getTemplatedDecl();
21   }
22   assert(FD);
23
24   if (SelectiveFriendAttr *Attr =
25           FD->getAttr<SelectiveFriendAttr>()) {
26     const Expr *E = Attr->getExpr();
27     const UnaryOperator *UO =
28         cast<UnaryOperator>(E);
29     const DeclRefExpr *DRef =
30         cast<DeclRefExpr>(UO->getSubExpr());
31     if (cast<NamedDecl>(DRef->getDecl()) !=
32         Target.getTargetDecl()) {
33       return AR_inaccessible;
34     }
35   }
36
37   return AR_accessible;
38 }
```

Figure 3.12: Constraint on selective friend attribute

access expression cannot have access to the member based on this specific friend declaration, therefore we return with `AR_inaccessible`; otherwise, we return with `AR_accessible`.

By calling the `SelectiveFriendAttr()` function, we increase the complexity of the compilation process. To estimate the overhead, we did some measurements on the modified compiler. We generated a file which contains friend declarations and member access expressions. Let n denote the number of member accesses and x denote the number of friend declarations. For instance, one generated file has the following form:

```
class A {
  int a;
  void mem_fun() {}
  friend void friend_fun0() {} // 1st
  friend void friend_fun1() {} // 2nd
  // ...
  friend void friend_funX() {} // xth
  friend void caller(A &a);
};
void caller(A &a) {
  [&a]() { a.mem_fun(); }(); // 1st
  [&a]() { a.mem_fun(); }(); // 2nd
  // ...
  [&a]() { a.mem_fun(); }(); // nth
}
```

In the case of selective friends `class A` has a different form:

```
class A {
  int a;
  void mem_fun() {}
  __attribute__((friend_for(&A::a))) friend void friend_fun0() {} // 1st
  __attribute__((friend_for(&A::a))) friend void friend_fun1() {} // 2nd
  // ...
  __attribute__((friend_for(&A::a))) friend void friend_funX() {} // xth
  friend void caller(A &a);
};
```

We compiled the generated files several times, so we could measure the compilation time in the domain of n and x. We executed the compiler on an Intel(R) Core(TM) i7 class processor. Let us bind x to be a constant 100. Figure 3.13 displays the performance of the compiler, when the number of the friend functions is a constant (exactly 100) and the number of the member accesses is growing. `baseline` denotes the performance of the compiler without any modification, the `new` refers to the compiler with the modification in it and the `selective` denotes the performance of the modified compiler when we have the attached selective attributes to the friend declarations. Our conclusion is that the compile time of selective friends scarcely depend on the number of member access expressions when the number of friend declarations is less than a hundred. However, when we fix n to $1000(N)$

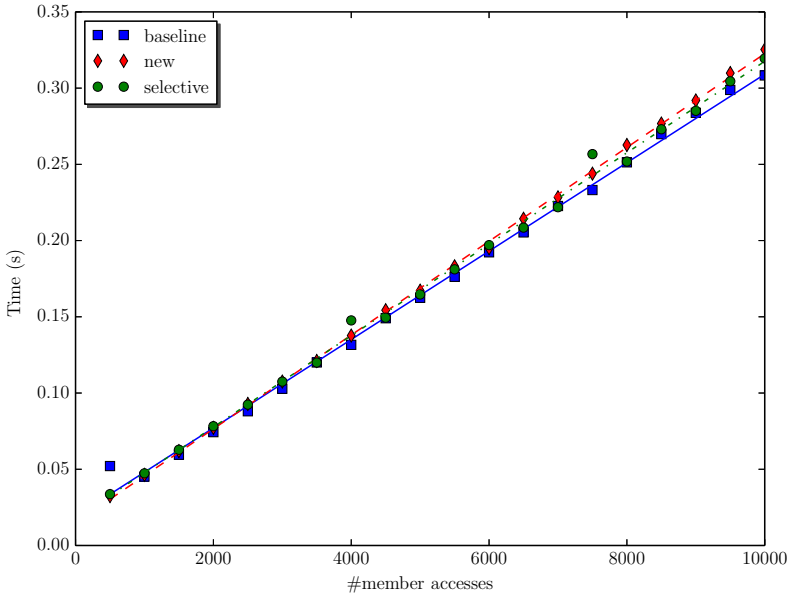


Figure 3.13: Comparing compile times, #friends: 100

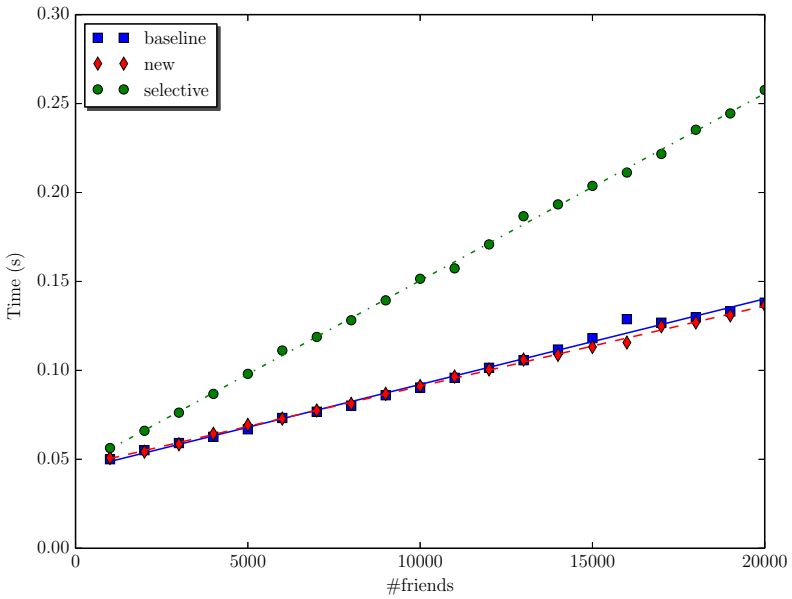


Figure 3.14: Comparing compile times, #members: 1000

and let the number of the friend function declarations grow gradually, we get the results in Figure 3.14. Let $f(x)$ denote the compile time of N member accesses (`baseline`). Let $g(x)$ denote the compile time of N member accesses in the domain of the number of the friend declarations where those declarations are selective friends (`selective`). We can see a correlation between $f(x)$ and $g(x)$: $g(x) = C(x) * f(x), 1.12 < C(x) < 1.87, 1000 <= x <= 20000$. We state the compilation overhead is always less than 3x if the number of friend declarations is less than 20000.

We conclude that the performance of the compiler is indeed degraded, but we could measure noticeable compile time overhead only when the number of the friend declarations were above a hundred. (Note, we estimate that having more than a hundred friend declarations in a class is quite a rare case in C++ source codes.) The measurement scripts and the resulted files from which we generated the charts are publicly available online [20].

3.5.2 Eiffel like Syntax

In our implementation, we use expressions as attribute arguments: we are taking the address of a member variable. If we did the same with a member function that would certainly require the compiler to generate the executable code of that function to be able to take its address. As a result, an otherwise non-generated (inlined) function is generated into the final executable. Also, in order to be able to take the address of a `static const` integral member variable, we need to define that variable [62, page 210–211]. So we would need to define it just because of the selective friend declaration. These side effects might not be wanted in some cases.

With our implementation, we have to specify a valid expression as an argument to the attribute. Therefore, we cannot restrict access to a specific member type. For that we would need to have the attribute definition to look like this:

```
def SelectiveFriend : InheritableAttr {
  // ...
  let Args = [TypeArgument<"MemberType">];
  // ...
}
```

This is one limitation of the implementation of the attributes in the used Clang compiler, we cannot specify an argument which is either a type or an expression. We have to make the decision during the design of the attribute.

Because of the described difficulties, we might want to seek other alternatives. One such alternative syntax could be to annotate each and every member via an attribute (Figure 3.15). The current C++ standard (C++17) enables us to tag the friend functions/classes which may access the private members. However, with the alternative approach presented in Figure 3.15 we would have to tag the members

```
void func(A &a);
class A {
  [[friend_for (func)]] int x = 0;
  int y = 0;
};

void func(A &a) {
  a.x = 1; // OK
  a.y = 1; // expected-error,
           // {{'y' is a private member of 'A'}}
}
```

Figure 3.15: Alternative syntax: annotate the members to provide access for a function

to indicate if they are accessible outside of the class. Actually, this method is very similar to Eiffel's access control.

The Eiffel programming language uses a special tagging mechanism for every class member to achieve selective friend like access control. Eiffel *features* are synonyms to C++ member variables and member functions. A feature can be either a field or a method [183, 184]. Features can be exported to all classes [156]:

- `feature {ANY}` means that the specific feature is available to all classes (analogous to C++ public).
- `feature {NONE}` means that the specific feature is not available to any classes (analogous to C++ private).
- `feature {CLASS_A, CLASS_B, CLASS_C}` means the features will be accessible by all three classes and in all of their proper descendant classes, but not in any other classes.

3.6 Related Work

3.6.1 Private Usage of Friend Classes

In 2005, English, Buckley and Cahill defined a number of software metrics that measure the extent to which friend class relationships are actually used in systems [185]. Our experiments confirm these earlier results and extend them. They defined the following relevant metrics:

- Actual Friend Methods (AFM): Counts the number of methods in a friend class that access hidden members of the befriending class.

- Actual Friend Classes (AFC): The number of friend classes in a software system which are actually exploited through friendship, i.e. the number of those classes which have $AFM >= 1$. If we divide the AFC value of a system with the number of friend classes in it we may have percentage value about friend classes declared as friends superfluously.

- Complexity in the Forward Direction for Friends (CCFF(1)): Counts the number of distinct interactions of a friend class with hidden methods and attributes of the befriending class. (CCFF(1) is a refinement of the CCF(1) metric of Wilkie and Kitchenham [186].)

- Response set For Friend Class (RFFC(1)): In a friend class, it counts the number of distinct hidden members which are accessed in the befriending class.

- Coupling Complexity in the Backward direction for Friends (CBOF(Back)): Counts the number of declared friend classes which actually access hidden members of the class, i.e. that utilize the friend declaration.

- Actual Friend Class Relationships (AFCR): This is a system level metric, like AFC. AFCR is the total sum of all the friend class relationships actually exploited in a system. Therefore, it is the sum of CBOF(Back) for all the classes in the system.

- Members Accessed by Friends (MAF): Counts the number of hidden members of a befriending class which are accessed by its friend classes.

All of these metrics are interpreted only in the context of friend classes, they are not defined for friend functions. Also, it is not clear how primary class templates, template instantiations and specializations affect these metrics. Our metrics of private usage (and private usage ratio) is applicable to friend functions and member functions of friend classes as well. Also, in our empirical study, we handle class templates, their specializations and instantiations with a well-defined algorithm as described in section 3.4.

English and Buckley have evaluated the above metrics on a number of open-source projects. For the systems in their study, the AFC percentage ranged from 42.8% to 100%. For about 60% of all systems, the percentage of friend classes which exploit some of the friendship available to them was less than 75%. Therefore, for more than half of the systems in their study at least 25% of friend class declarations were shown to be redundant. The CBOF(Back) metric returned mostly small values. There were just 2 systems with a median value greater than 1 for CBOF(Back). In addition, the maximum value of CBOF(Back) was less than 9 for all but one system. This again indicated only a small subset of friend classes

accessed hidden members. For the systems they analyzed, a large proportion of befriending classes returned a zero value for MAF, indicating that a considerable number of friend declarations were not exploited and as a result, no hidden members of the befriending classes were accessed directly. In many systems a large proportion of classes had an RFFC(1) value of zero, indicating that no use was made of any friend relationship, where the class was declared as a friend. English and Buckley concluded there are many friend classes which do not exploit friendship. We confirm their conclusion based on our measurement results: The number of friend class function instances with zero private entity usage is quite high in the measured four libraries (Figure 3.9). With other words, there are many erroneous or superfluous friend declaration of friend classes.

They also concluded that there are many friend class declarations where friend function declarations might be more appropriate. This conclusion is also aligned with our empirical results in Figure 3.9 and 3.7. We can see that most of the friend class function instances have zero private usage. In the case of friend functions the ratio of incorrect/correct friend function instances is significantly lower than the ratio in the case of friend classes, thus friend function declarations are more appropriate.

There are some other results in English's study which support the raison d'etre of selective friends. The median value for the AFM metric was found to be 1 for many systems in their research. None of the 28 systems had a median value greater than 3. For 12 of the 28 systems, the median value of CCFF(1) was <= 3. They measured that in almost 50% of systems, 50% of the classes declared as friends were involved in less than or equal to 3 interactions that were dependent on the friend construct. Also, in all but 2 systems the median value of MAF is less than or equal to 3. The median value of RFFC(1) for 26 of the 28 systems was less than or equal to 3. Therefore, in almost all systems, 50% of classes declared as friends required access to less than or equal to 3 hidden members in other classes. These numbers are aligned with our observation that the vast majority of the correct friend class function instances access only 1-4 private entities (Figure 3.9). English and Buckley concluded that for all systems at least 50% of classes declaring friends only access a small number of hidden members. We confirm this statement based on our metrics on private usage ratio of friend class function instances (Figure 3.11). They also concluded that the level of protection assigned to some class members should be reconsidered, especially where friend classes only access a small number of hidden members in a class. This again confirms the need for a selective friend construct.

3.6.2 Friends and Inheritance

Counsell and Newson studied a number of hypotheses about relationships between the use of friends and other internal attributes of a class [187]. Their results suggest that the friend mechanism is used as an alternative to inheritance. They examined four C++ systems of varying sizes and analysed data related to friends collected for each. The following five hypotheses were investigated:

- The more friends in a class, the less (non-friend) coupling found in that class.
- The more friends in a class, the more methods found in that class.
- Classes declared as friends of other classes have less inheritance than other system classes.
- Classes containing friends that engage in inheritance have fewer descendants than other system classes.
- Classes that do not engage in any inheritance have more friends than classes which do engage in inheritance.

Their results showed a lack of statistical significance with any of the class metrics collected. No evidence was found to support the hypothesis that classes declared as friends of other classes used inheritance any less than other classes. Strong evidence was found, however, to support the hypothesis that, firstly, friends tended to be found in classes deep in the inheritance hierarchy; secondly, that classes not engaging in inheritance use friends considerably more than classes that do. Their empirical evidence also suggests that friends are used primarily as a means for facilitating operator overloading.

English, Buckley and Cahill replicated and refined the empirical study made by Counsell and Newson [188, 189, 190, 191]. They refined their measurements by excluding friend constructs which participate in operator overloading. Counsell and Newson handled friendship symmetrically. However, classes or functions declared as friends have the potential to import additional functionality from classes which declare this class as a friend. Similarly, a class which declares friends has the potential to export more functionality to the classes or functions that it declares as friends. Therefore the coupling of classes is affected. Consequently, English and Buckley handled coupling metrics asymmetrically for the befriending classes and for the friend functions/classes.

Counsell and Newson included all overloaded operators (friend functions) in their measurements. English and Buckley excluded all overloaded friend operator functions. We neither exclude nor include all overloaded operators: we do not distinguish overloaded operators from regular functions. Although, we further

refine and analyse the results with the concept of Meyers candidates (defined in Section 3.4.1). Meyers candidates are those friend functions which are using friendship not because of accessing private members but for other technical reasons (like automatic function template instantiation). An overloaded operator may or may not be a Meyers candidate, just as any other regular function. Meyers candidates influence our measurement results: We exclude Meyers candidates in the calculation of the private usage ratio. We do not tag friend functions as being erroneously friends if they are Meyers candidates.

3.6.3 Alternatives for Selective Friends

With current C++ we can achieve semantically similar effects to the proposed selective friend language construct. However, they all have their own drawbacks and difficulties.

The Access Key Idiom

First, there is no such acknowledged pattern which has this name. Some people tend to refer the presented methodology as key-oriented access protection, passkey idiom [192], access key idiom or higher-order friendship [193]. We refer to this as the *access key idiom*. Figure 3.16 presents an example for the idiom. The `Owner` class has a privileged function, of which we want to restrict the access. This is achieved by the fact that this function requires a `Key` object to be passed as an argument. Since `Key` has a private constructor only its friends can construct any object from it. Therefore `Key` controls which classes and functions can have access to the privileged function. With this idiom we can provide selective friendship: we can control access to a specific member function and we can restrict access of friends to a set of private methods. (`Owner::foo` cannot be accessed by anybody). The drawback of this pattern is that we cannot handle the accessing of member fields with it, we can handle only member functions.

The Attorney-Client Idiom

This pattern uses an auxiliary class to handle (restrict) access to the private members of a specific class. Figure 3.17 demonstrates this intermediate class is the `Attorney`; just as in real life it protects its `Client`, so other entities can access it only via the lawyer [194]: In this example, the `Bar` class has limited access to `Client`, it can call only `Client::A()` via `Attorney::callA()`. So `Bar` acts as a selective friend of class `Client`. Also, the `Attorney` controls which classes can have access to the internals of `Client`. `Attorney` has a similar role as `Key` has in the access key idiom. The use of this idiom does not scale well, because we

```
class Key;

class Owner {
public:
  void privileged(const Key &) {}
private:
  void foo();
};
class Key {
  friend class User;
  friend void userFunc(Owner &owner);
private:
  Key() {}
};
class User {
public:
  void usePrivileged(Owner &owner) {
    owner.privileged(Key{}); // OK
  }
};
void userFunc(Owner &owner) {
  owner.privileged(Key{}); // OK
}
void noAccess(Owner &owner) {
  owner.privileged(Key{}); // compile error
}
```

Figure 3.16: Example of the access key idiom

```
class Client
{
private:
  void A(int a);
  void B(float b);
  void C(double c);
  friend class Attorney;
};

class Attorney {
private:
  static void callA(Client & c, int a) {
    c.A(a);
  }
  friend class Bar;
};

class Bar { /* ... */ };
```

Figure 3.17: Example of the attorney-client idiom

would need to define several attorney classes if we wanted to provide access for the different combination of members. Also, using too many attorneys might result in unmaintainable and hardly understandable code.

3.7 Future Research

It is an important candidate for future work to create a selective friend implementation without the current limitations. For instance, we could enhance the implementation for selective friend classes, or we could further develop it to support more than one selected member. Also, we are planning to implement this new language element without attributes.

It is worth to investigate the possibilities of `const` qualified selective friends. For instance, consider the following code block:

```
class A {
  int x = 0;
  int y = 0;
  friend for (x) B const; // read-only
};
void B::func(A &a) { // expected-error, B has const access only
  int i = a.x;
  a.x = 1; // or expected-error here,
           // a is implicitly casted to 'const A&'
}
```

The member functions of `B` could access only the listed members of `A` and every instance of `A` would be handled as a `const` object.

3.8 Conclusion

We executed a fine-grained measurement about the usage of friends on several open source projects. Based on these empirical results we claim that friendship in C++ is often misused. In a number of cases, friend functions access only public members or not access members at all. In other cases, they gain superfluous access to private members, which is a possible source of errors. Current C++ compilers and static analysis tools do not issue a warning on suspicious friend usage. However, with our publicly available friend inspection tool, we can list the possible erroneous uses of friend declarations.

The current C++ language specification does not allow to restrict the access of private members. With selective friends, we can establish a more sophisticated access control. Our proposal may decrease the degradation of encapsulation. We created a proof-of-concept implementation based on the LLVM/Clang compiler infrastructure to show that such constructs can be established with a minimal

syntactical and compilation overhead. Even without any change to the current C++ standard, the attribute-based `friend_for` implementation could be useful. The compilers that support it could provide proper diagnostics when the semantics of the attribute is violated. The tools which do not support it would just simply ignore the `friend_for` specification as unknown attributes must be ignored according to C++17.

3.9 Contribution

Thesis 3 (Selective friend)**.** *I have investigated how the friendship mechanism is used in C++ programs. I have shown that there are various holes and errors in friend usage like friend functions accessing only public members or not accessing members at all. I have proposed a selective friend language construct for C++ which can restrict friendship only to well-defined members. I have demonstrated that such a new language element may decrease the degradation of encapsulation and significantly increase the diagnostic capacity of the compiler. To underpin my statements, I have made a publicly available friend inspection tool, which can list the possible erroneous uses of friend declarations and I have created a proof-of-concept implementation of selective friends based on the LLVM/Clang compiler infrastructure.*

thesis name	relevant publications									
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
(1) New non-intrusive testing methods	∘	∘	∘	∘					∘	
(2) Extending access for non-intrusive and white-box testing	∘							∘		∘
(3) **Selective friend**					•					
(4) High-level abstraction for the read-copy-update pattern						∘	∘			

Chapter 4

The Read-Copy-Update Pattern

Concurrent programming with classical mutex/lock techniques does not scale well when reads are way more frequent than writes. Such situation happens in operating system kernels among other performance critical multithreaded applications. Read – copy – update (RCU) is a well know technique for solving the problem. RCU guarantees minimal overhead for read operations and allows them to occur concurrently with write operations. RCU is a favourite concurrent pattern in low-level, performance critical applications, like the Linux kernel. Currently there is no high-level abstraction for RCU for the C++ programming language. In this section, we present our C++ RCU class library to support efficient concurrent programming for the read-copy-update pattern. The library has been carefully designed to optimise performance in a heavily multithreaded environment, at the same time providing high-level abstractions, like smart pointers and other C++11/14/17 features.

4.1 Context and Motivation

Read-copy-update is a concurrent design pattern [195, 196] which allows extremely low run-time overhead for readers. Updates can happen concurrently with reads as they leave the old versions of the data structure intact; this way the pre-existing readers can finish their work. Updates might require more overhead than reads and their effect might be delayed.

An example sequence of reads and update operations is depicted in Figure 4.1. First, three reader threads access the data concurrently (4.1a, 4.1b, 4.1c). Then comes an updater, which copies the data. During the copy the updater behaves as a reader of the data, thus it can operate concurrently with the existing readers (4.1d). The update operation is performed on the copied data. While the update is ongoing, pre-existing readers may finish their work (4.1e), also any new reader

will get a reference to the old data (4.1f). Once the updater had finished its work then new readers will obtain a reference to the new data which had been previously copied and modified by the updater (4.1g, 4.1h). Any subsequent new reader will access the updated data (4.1i). Once all the pre-existing readers of the old data finish their work then the old data is released (4.1j, 4.1k, 4.1l, 4.1m).

In contrast to readers-writers lock [197], RCU does not block the writers if there are concurrent readers. Figure 4.2 describes the differences between the two mechanisms. As described in Paul Mckenney's Perfbook [39], "once the update is received, the rwlock writer cannot proceed until the last reader completes, and subsequent readers cannot proceed until the writer completes. However, these subsequent readers are guaranteed to see the new value, as indicated by the green shading of the rightmost boxes. In contrast, RCU readers and updaters do not block each other, which permits the RCU readers to see the updated values sooner. Of course, because their execution overlaps that of the RCU updater, all of the RCU readers might well see updated values, including the three readers that started before the update. Nevertheless only the green-shaded rightmost RCU readers are guaranteed to see the updated values."

Classical RCU first appeared in the Linux kernel in 2002 [198, 39]. It provides the following reader side primitives: `rcu_read_lock()` and `rcu_read_unlock()`. Read-side critical sections may use `rcu_dereference()` to access RCU protected pointers.

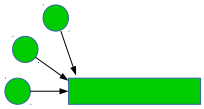
On the update side we may use the `synchronize_rcu()` primitive and `rcu_assign_pointer()` to assign values to protected pointer. Pointers stored by `rcu_assign_pointer()` can be fetched from within read-side critical sections by `rcu_dereference()`.

The pseudo code in Figure 4.3 demonstrates how these primitives can be used to implement the lookup and remove operations on a simple linked list of key-value pairs. This implementation is a simplified excerpt of McKenney's pre-BSD routing table example [39]. With `rcu_read_lock()` and `rcu_read_unlock()` we indicate the reader side critical section. In this read-side critical section we traverse through the list (`find()`) and once we found the key we return with the associated value. In the implementation of `find()` we have to use `rcu_dereference()` to access the elements in the list. It might happen that the key is not in the list, in that case, we again close the critical section and then return with a special value indicating the element is not in the list.

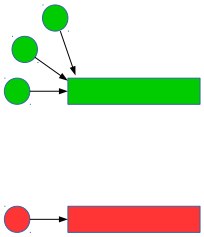
In `remove()` we have to use a spin lock in order to protect the list from concurrent write operations. The block which is protected by the spin lock is the write-side critical section. We iterate over the list trying to find the key and if we found it then we unlink (`remove_node()`) it from the list. In the realization of the `remove_node()` we have to use the `rcu_assign_pointer()` primitive. After

(a) First reader

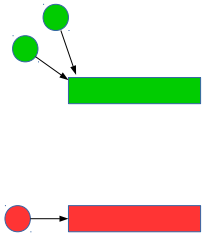
(b) Second reader



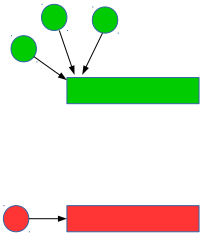
(c) Third reader



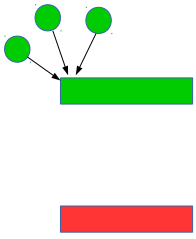
(d) Updater, creates a copy and modifies that



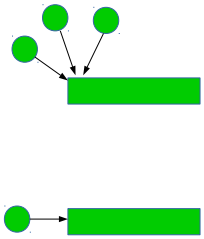
(e) Pre-existing reader finishes



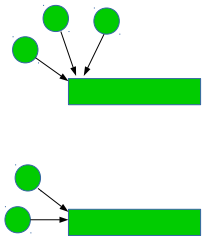
(f) New reader, references the old data since update is ongoing



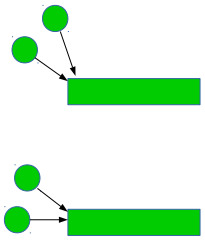
(g) Updater finishes



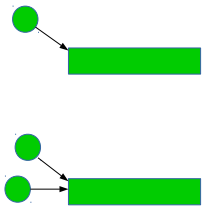
(h) New reader, references the updated data



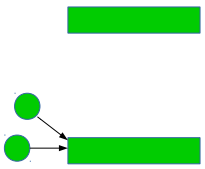
(i) Another new reader, references the updated data



(j) One reader which references the old data finishes



(k) Another reader which references the old data finishes



(l) Last reader which references the old data finishes

(m) Old data is freed

Figure 4.1: Example sequence of read and update operations in RCU

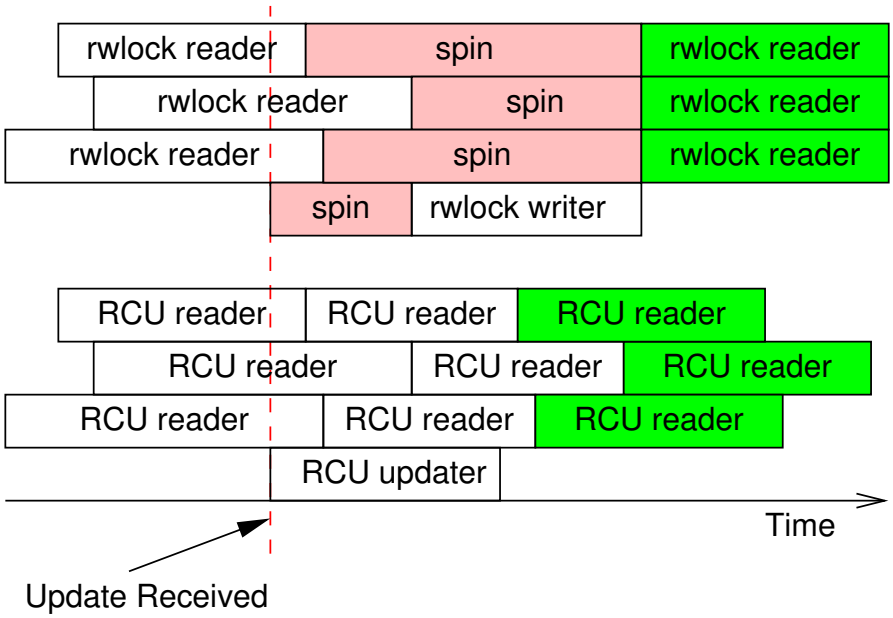


image source: Paul McKenny, Perfbook [39]

Figure 4.2: RCU and readers-writer lock comparison

the removal, with the `synchronize_rcu()` primitive we wait for all pre-existing RCU read-side critical sections to completely finish. Then we can deallocate the list node which is no longer needed and this way can close the write-side critical section by releasing the lock.

Classic RCU requires that read-side critical sections obey the same rules obeyed by the critical sections of pure spinlocks: blocking or sleeping of any sort is strictly prohibited. Since 2002 many different RCU flavours have appeared in the Linux kernel which relaxes this strict requirement. Using realtime RCU [199, 200, 201] read-side critical sections may be preempted and may block while acquiring spinlocks. Sleepable RCU allows more, it permits arbitrary sleeping (or blocking) within RCU read-side critical sections [202, 203].

The different RCU flavours in the Linux kernel are naturally dependent on the kernel internals, for example on the scheduler. Obviously, they cannot be used in user space. Userspace RCU (URCU) [204, 205] was created by Desnoyers in 2009 and has a similar API to the kernel space RCU flavours. URCU has different variants and implementations. For instance, the Quiescent-State-Based Reclamation RCU (QSBR) provides near-zero read-side overhead but the price of minimal overhead is that each thread in an application is required to periodically invoke `rcu_quiescent_state()` to announce that it resides in a quiescent state [206]. The general-purpose user space realization can be used in applications

```
SPINLOCK(lock);

Value lookup(List list, Key key) {
  Node* node;
  Value local_value;
  rcu_read_lock();
  // iterate over the list and return the value
  // of the found element
  if (node = find(list, key)) {
    local_value = node->value
    rcu_read_unlock();
    return local_value;
  }
  rcu_read_unlock();
  return not_found;
}

void remove (List list, Key key) {
  Node* node;
  spin_lock(lock);
  // iterate over the list and find the key
  if (node = find(list, key)) {
    remove_node(list, node);
    spin_unlock(lock);
    synchronize_rcu();
    free(node);
    return;
  }
  spin_unlock(lock);
}
```

Figure 4.3: Usage of RCU in a linked list

```
class X {
  std::vector<int> v;
  mutable std::mutex m;

public:
  int sum() const { // read operation
    std::lock_guard<std::mutex> lock{m};
    return std::accumulate(v.begin(), v.end(), 0);
  }
  void add(int i) { // write operation
    std::lock_guard<std::mutex> lock{m};
    v.push_back(i);
  }
};
```

Figure 4.4: A shared collection

where we cannot guarantee that each thread will invoke `rcu_quiescent_state()` sufficiently often. However, this versatility has its own price, general-purpose RCU has to use memory barriers in the read-side. A third variant uses POSIX signals to eliminate these barriers, obviously, this flavour cannot be used on non-POSIX systems.

URCU has been proposed to be incorporated into the C and C++ standard with the C API provided by Desnoyers realization [207]. URCU provides a low-level C API, therefore it is more prone to errors in C++ programs than a well established high-level C++ API can be. For instance, it is easy to forget to call `rcu_read_unlock()` on all return paths. Also, the user must remember to manually lock a spinlock in the update operation. This is by itself a source of errors, but this lock must be released on all return paths, which is again another source of errors. Furthermore, in URCU there is no automatic memory reclamation; to deallocate memory, first, we have to use the `synchronize_rcu()` primitive. (Note that besides Desnoyers realization there are a surprisingly large number of other lesser known userspace RCU implementations, and more are being created all the time. E.g. [208, 209].)

In this chapter we present an alternative implementation for user space RCU as a C++ smart pointer, thus there is no need to manually deallocate memory. Our realization provides a high-level abstraction C++ API to the users, so they can use a simple construct which is not prone to errors, still its performance is satisfying for most of the use cases.

4.2 Towards a Higher Level Abstraction for RCU

Figure 4.4 presents a collection that is shared among multiple readers and writers in a concurrent manner. It is a common way to make the collection thread-safe by holding a lock until the iteration is finished (on the reader thread). This approach does not scale well, especially when reads are way more frequent than writes [39]. Instead of a simple `lock_guard` we could use a readers-writers lock [197], but that would scale badly as well, especially when we have multiple concurrent writers [39].

The first idea to make it better is to have a shared pointer and hold the lock only until that is copied by the reader or updated by the writer:

```
class X {
  std::shared_ptr<std::vector<int>> v;
  mutable std::mutex m;

public:
  X() : v(std::make_shared<std::vector<int>>()) {}
  int sum() const { // read operation
    std::shared_ptr<std::vector<int>> local_copy;
    {
      std::lock_guard<std::mutex> lock{m};
      local_copy = v;
    }
    // assume processing the data takes longer than copying it
    return std::accumulate(local_copy->begin(), local_copy->end(), 0);
  }
  void add(int i) { // write operation
    std::shared_ptr<std::vector<int>> local_copy;
    {
      std::lock_guard<std::mutex> lock{m};
      local_copy = v;
    }
    local_copy->push_back(i);
    {
      std::lock_guard<std::mutex> lock{m};
      v = local_copy;
    }
  }
};
```

Now we have a race on the pointee itself during the write. So we need to have a deep copy:

```
void add(int i) { // write operation
  std::shared_ptr<std::vector<int>> local_copy;
  {
    std::lock_guard<std::mutex> lock{m};
    local_copy = v;
  }
  auto local_deep_copy = std::make_shared<std::vector<int>>(*local_copy);
  local_deep_copy->push_back(i);
  {
    std::lock_guard<std::mutex> lock{m};
    v = local_deep_copy;
  }
}
```

The copy construction of the underlying data (`vector<int>`) is thread-safe since the copy constructor parameter is a constant reference to `vector<int>`.

Still, there is one more problem: if there are two concurrent write operations then we might miss one of them. We should check whether the other writer had done an update after the actual writer has loaded the local copy. If it did then we should load the data again and try to do the update again. This leads to the idea of using an `atomic_compare_exchange` in a while loop. We could use an `atomic_shared_ptr` if that was included in the current C++ standard, but until then we have to be satisfied with the free function overloads for `shared_ptr`:

```
1  class X {
2    std::shared_ptr<std::vector<int>> v;
3
4  public:
5    X() : v(std::make_shared<std::vector<int>>()) {}
6    int sum() const { // read operation
7      auto local_copy = std::atomic_load(&v);
8      return std::accumulate(local_copy->begin(), local_copy->end(), 0);
9    }
10   void add(int i) { // write operation
11     auto local_copy = std::atomic_load(&v);
12     auto exchange_result = false;
13     while (!exchange_result) {
14       // we need a deep copy
15       auto local_deep_copy =
16           std::make_shared<std::vector<int>>(*local_copy);
17       local_deep_copy->push_back(i);
18       exchange_result = std::atomic_compare_exchange_strong(
19           &v, &local_copy, local_deep_copy);
20     }
21   }
22 };
```

Figure 4.5: Using atomic shared pointer

These free function overloads take a simple `shared_ptr` as a parameter and perform the specific atomic operations:

```
template <class T>
std::shared_ptr<T> atomic_load(const std::shared_ptr<T>* p);

template <class T>
bool atomic_compare_exchange_strong(std::shared_ptr<T>* p,
                                    std::shared_ptr<T>* expected,
                                    std::shared_ptr<T> desired);
```

Note, the `atomic_shared_ptr` class template which would replace these free functions might be included in the C++20 standard [210].

In the write operation, we do the update on the copy of the original pointee (line 17 of Figure 4.5) and not on the pointee of the member. During both the read operation and the write operation we do not modify the pointee. Thus, the element type of the member `shared_ptr` can be changed to be a constant:

```
class X {
  std::shared_ptr<const std::vector<int>> v;
  // ...
};
```

We might notice that we can move-construct the third parameter of `atomic_compare_exchange_strong`, therefore we can spare a reference count increment and decrement:

```
exchange_result =
    std::atomic_compare_exchange_strong(
        &v, &local_copy, std::move(local_deep_copy));
```

Regarding the write operation, since we are already in a while loop we could replace `atomic_compare_exchange_strong` with `atomic_compare_exchange_weak`. That can result in a performance gain on some platforms [63, 211]. However, `atomic_compare_exchange_weak` can fail spuriously[1]. Consequently, we might do the deep copy more often than needed if we used the weak counterpart.

In the current form of class `X` nothing stops another programmer (e.g. a naive maintainer of the code years later) to add a new reader operation, like this:

```
int another_sum() const {
  return std::accumulate(v->begin(), v->end(), 0);
}
```

This is definitely a race condition and a problem. To avoid this user error and to hide the sensitive technical details we created a smart pointer which we named as `rcu_ptr`. This smart pointer provides a general higher level abstraction above `atomic_shared_ptr`. Below we present how can we use `rcu_ptr` in our running example:

[1]Spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g., load-locked store-conditional machines [63]

```
class X {
  rcu_ptr<std::vector<int>> v;

public:
  X() : v(std::make_shared<std::vector<int>>()) {}
  int sum() const { // read operation
    std::shared_ptr<const std::vector<int>> local_copy = v.read();
    return std::accumulate(local_copy->begin(), local_copy->end(), 0);
  }
  void add(int i) { // write operation
    v.copy_update([i](std::vector<int> *copy) { copy->push_back(i); });
  }
};
```

The `read()` method of `rcu_ptr` returns a `shared_ptr<const T>` by value, therefore it is thread-safe. The existence of the `shared_ptr` in the scope enforces that the read object will live at least until this read operation finishes. By using the shared pointer this way, we are free from the ABA problem [212, 213] since the memory address associated with the object cannot be reused until the object itself is reclaimed [214]. The `copy_update()` method receives a lambda. This lambda is called whenever an update needs to be done, i.e. it will be called continuously until the update is successful. The lambda receives a `T*` for the copy of the actual data. We can modify the copy of the actual data inside the lambda.

4.3 Smart Pointer for RCU Semantics

In Figure 4.6 we present the simplified implementation of the `rcu_ptr` class template. The complete implementation is available and free to use at [21]. We provide a default constructor and a default destructor (lines 6 and 7). The move and copy operations are deleted (lines 9-12) because `rcu_ptr` is essentially a wrapper around an atomic type (we plan to support `atomic_shared_ptr` as soon as it is included in the standard). And all atomic types are neither copyable nor movable (because there is no sense to assign meaning for an operation spanning two separately atomic objects) [215, 216].

We can create an `rcu_ptr` from an lvalue or rvalue reference of `shared_ptr<const T>` (lines 14-15). These functions just simply copy or move their parameter into the member `shared_ptr`. There is no need to make these constructors thread-safe, because the construction can be done only by one thread.

Lines 21-27 is the realization of the `reset()` methods which receive a `shared_ptr<const T>` as an lvalue or rvalue reference parameter. We can use it to reset the wrapped data to a new value independent from the old value (e.g. `vector.clear()`). Actually, with the parameter, we overwrite the currently contained `shared_ptr`. The overwrite has to be an atomic operation in order to

```
template <typename T>
class rcu_ptr {
  std::shared_ptr<const T> sp;

public:
  rcu_ptr() = default;
  ~rcu_ptr() = default;

  rcu_ptr(const rcu_ptr& rhs) = delete;
  rcu_ptr& operator=(const rcu_ptr& rhs) = delete;
  rcu_ptr(rcu_ptr&&) = delete;
  rcu_ptr& operator=(rcu_ptr&&) = delete;

  rcu_ptr(const std::shared_ptr<const T>& sp_) : sp(sp_) {}
  rcu_ptr(std::shared_ptr<const T>&& sp_) : sp(std::move(sp_)) {}

  std::shared_ptr<const T> read() const {
    return std::atomic_load_explicit(&sp, std::memory_order_consume);
  }

  void reset(const std::shared_ptr<const T>& r) {
    std::atomic_store_explicit(&sp, r, std::memory_order_release);
  }
  void reset(std::shared_ptr<const T>&& r) {
    std::atomic_store_explicit(&sp, std::move(r),
                               std::memory_order_release);
  }

  template <typename R>
  void copy_update(R&& fun) {

    std::shared_ptr<const T> sp_l =
        std::atomic_load_explicit(&sp, std::memory_order_consume);

    std::shared_ptr<T> r;
    do {
      if (sp_l) {
        // deep copy
        r = std::make_shared<T>(*sp_l);
      }

      // update
      std::forward<R>(fun)(r.get());

    } while (!std::atomic_compare_exchange_strong_explicit(
                  &sp, &sp_l, std::shared_ptr<const T>(std::move(r)),
                  std::memory_order_release, std::memory_order_consume));
  }
};
```

Figure 4.6: The rcu_ptr class template

protect the member from concurrent `reset()` calls.

In lines 17-19, the `read()` method atomically loads the member `shared_ptr` and returns with a copy of that. The `copy_update()` function template (lines 29-48) receives an rvalue reference to an instance of a callable type. First, we create a local copy of the member as `sp_l` (lines 32-33). If this local copy is set (i.e the `rcu_ptr` instance is initialized) then we create a deep copy, that is we copy the pointee itself and we create a new `shared_ptr<T>` (denoted as `r`) pointing to the copy (lines 37-40). Note, that this is a non-constant shared pointer. In line 43 we call the callable and we pass a non-constant pointer to the new copy as a parameter. Then in lines 45-47 we exchange the member shared pointer with a `shared_ptr` to the deep copy if we find that the member still points to the same object of which we created the copy. If it turns out that is not the case (i.e. another thread was faster), then we repeat the whole deep copy update sequence until we succeed (line 36). The callers of the `copy_update()` function must be aware that in case of an unset (or default initialized) `rcu_ptr` the callable will be called with a null pointer as an argument. Also, a call expression with this function is invalid, if the wrapped data type (`T`) is a non-copyable type.

4.3.1 Memory Ordering

A `memory_order_release` store is said to *synchronize with* a `memory_order_acquire` load if that load returns the value stored or in some special cases, some later value [217, 63]. When a `memory_order_release` store synchronizes with a `memory_order_acquire` load, any memory reference preceding the `memory_order_release` store will *happen before* any memory reference following the `memory_order_acquire` load [217, 63]. This property allows a linked structure to be locklessly traversed by using `memory_order_release` stores when updating pointers to reference new data elements and by using `memory_order_acquire` loads when loading pointers while locklessly traversing the data structure [217]. A `memory_order_release` store is *dependency ordered before* a `memory_order_consume` load when that load returns the value stored, or in some special cases, some later value [217, 63]. Then, if the load carries a dependency to some later memory reference, any memory reference preceding the `memory_order_release` store will happen before that later memory reference [217, 63]. This means that when there is dependency ordering, `memory_order_consume` gives the same guarantees that `memory_order_acquire` does, but possibly at a lower cost [217].

In the classical RCU, the `rcu_dereference()` primitive implements the notion of a dependency ordered load, which suppresses aggressive code-motion compiler optimizations and generates a simple load on any system other than DEC Alpha, where it generates a load followed by a memory-barrier instruction. The

`rcu_assign_pointer()` primitive implements the notion of store release, which on sequentially consistent and total-store-ordered systems compiles to a simple assignment [204].

In our implementation of `rcu_ptr::copy_update()` function we can also use the release and consume semantics. We cannot use relaxed ordering because in case of that if the `fun` is inlined and `fun` itself is not an ordering operation or it does not contain any fences then the load or the compare_exchange might be reordered into the middle of `fun`. Also, we need to "see" the latest updates so we can copy and update the "most recent" version. Though, there is a data dependency chain: `sp_l->r->compare_exchange(...,r)`. So if all the architectures were preserving data-dependency ordering, then we would be fine with relaxed. However, some architectures do not preserve data-dependency ordering (e.g. DEC Alpha), therefore we need to explicitly state that we rely on that neither the CPU nor the compiler will reorder data dependent operations. This is what we express with the consume-release semantics. Consequently, during all the atomic load operations in the `rcu_ptr` class template, we can use `memory_order_consume` and during all atomic store operations (including the read-modify-write operation) we use `memory_order_release`. If the definition of the `fun` callable is unseen by the compiler (i.e. it is defined in another translation unit) then the user has to annotate the declaration of the callable with the `[[carries_dependency]]` attribute [63]. Otherwise, the compiler may assume that the dependency chain is broken during the call and consequently it would fall back to the safer but less efficient acquire semantics [63].

Unfortunately, the consume memory order is temporarily deprecated in C++17. It is widely accepted that the current definition of `memory_order_consume` in the C++11/14 standard is not useful. All current compilers essentially map it to `memory_order_acquire`. The difficulties appear to stem both from the high implementation complexity and from the fact that the current definition uses a fairly general definition of "dependency" [218, 217]. As such, the consume ordering has to be redefined. While this work is in progress, hopefully ready for the next revision of C++, users are encouraged to not use this ordering and instead use acquire ordering, so as to not be exposed to a breaking change in the future. As for our `rcu_ptr`, in order to reach the consume semantics, we may use hardware specific instructions in the future to overcome the mentioned problem.

4.3.2 Lock-Free atomic_shared_ptr

Our `rcu_ptr` can be used with the free functions overloads of the `atomic_` prefix [63, section 20.8.2.6] for `std::shared_ptr`. Since the `atomic_shared_ptr` [210] is still in experimental phase, we use our own wrapper template class around the free

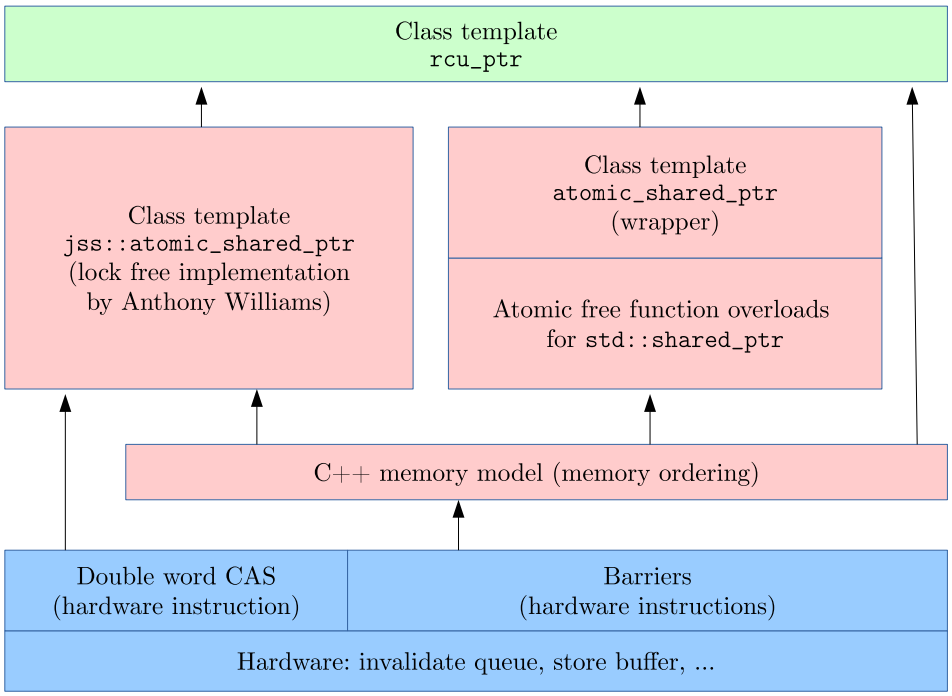


Figure 4.7: rcu_ptr and its dependencies

functions. The free functions are implemented in terms of a spinlock in the currently available standard libraries. Having a lock-free `atomic_shared_ptr` would be really beneficial. However, implementing a lock-free `atomic_shared_ptr` in a portable way can have extreme difficulties [219]. Though, it is easier on architectures where the double word CAS operation is available as a CPU instruction as we can see that with Anthony Williams implementation [220]. We can use Williams' implementation with our `rcu_ptr` class template as well if a double word CAS operation is available. We depict the relations between `rcu_ptr` and its dependent components in Figure 4.7.

4.4 Performance Evaluation

We executed performance measurements on a dual CPU system (two Intel® Xeon® X5670 CPUs). Each CPU had 6 physical cores with hyper-threading enabled, this

sums up to 24 threads. Also, each CPU had 12MB cache. We used Ubuntu 14.04 operating system (Linux kernel 3.13).

We took the class `X` from the running example (presented in Figure 4.4) and slightly changed it:

```
class X {
  std::vector<int> v;
  const int default_value = 1;
  mutable std::mutex m;

public:
  X(size_t vec_size) : v(vec_size, default_value) {}
  int read_one(unsigned index) const { // read operation
    std::lock_guard<std::mutex> lock{m};
    return v[index];
  }
  void update_all(int value) { // write operation
    std::lock_guard<std::mutex> lock{m};
    for (auto& e : v)
      e = value;
  }
};
```

We added a constructor via which we can set up the size of the `vector`. We modified the read operation to read only one value from the vector. We also changed the write operation to update all elements in the vector. We implemented this modified class in terms of several different synchronization mechanisms:

- **std mutex**. Standard mutex from the C++ Standard Template Library (STL). We used the STL implementation `libstdc++` from GNU Compiler Collection (version 5.4). On POSIX systems, `std::mutex` uses `pthread_mutex_lock` and `pthread_mutex_unlock` functions from the `pthread` library. On Linux, these `pthread` functions are implemented in terms of `futex` (fast userspace mutex) [221] system call. It provides very fast uncontended lock acquisition and release. The futex state is stored in a user-space variable. Atomic operations are used in order to change the state of the futex in the uncontended case without the overhead of a syscall. In the contended cases, the kernel is invoked to put tasks to sleep and wake them up.

- **tbb qrw mutex**. Intel® TBB queuing reader-writer mutex [222]. A `queuing_rw_mutex` is scalable, in the sense that if a thread has to wait to acquire the mutex, it spins on its own local cache line. A `queuing_rw_mutex` is fair. Threads acquire a lock on a `queuing_rw_mutex` in the order that they request it.

- **tbb srw mutex**.Intel® TBB spin reader-writer mutex [222]. A `spin_rw_mutex` is not scalable or fair. It is ideal when the lock is lightly

contended and is held for only a few machine instructions. If a thread has to wait to acquire a `spin_rw_mutex`, it busy waits, which can degrade system performance if the wait is long. However, if the wait is typically short, a `spin_rw_mutex` significantly improves performance compared to other mutexes.

- **rcuptr**. Our `rcu_ptr` with non-lock-free atomic shared pointer. We use a wrapper template class which encapsulates the free function overloads for atomic operations on a standard `shared_ptr`.
- **rcuptr jss**. Our `rcu_ptr` with Anthony Williams' lock-free atomic shared pointer [220]. Note that the examined Intel CPU has the double word CAS operation.
- **urcu bp**. The bulletproof version of the URCU library. We used the bulletproof version because that is the general version of URCU. The "bulletproof" version is the only one which can be used even when we cannot register individual threads with the URCU library.

We created a separate test binary for each mechanism. Each test binary consists of a timer thread which ticks approximately after one second, one writer thread and several reader threads (configurable number). As for the measure metrics, we count how many times a reader or writer thread finishes its operation during the elapsed time period. The timer thread sets an atomic stop flag while all the other threads read this flag continuously and they stop when it is set. We used relaxed memory ordering for writing and reading this flag in order to make sure that the cache system is not affected by the measurement itself. We executed each test binary with a different number of reader threads and with different vector sizes. We executed one test binary with a specific configuration (number of threads, vector size) five times. During the evaluation of each performance indicator value, we dropped the smallest and the largest values and we took the average of the remaining three values. The measurement scripts and the source code for the test binaries are readily available at [21], thus our measurements are easily replicable on any other hardware.

We experienced that if the size of the vector is really small (smaller than 4KiB) then the read-side performance of the RCU mechanisms are outperformed by a simple standard mutex. However, as the data grows, the RCU mechanism is getting an advantage over the standard mutex and over the read-write mutexes (Figure 4.8). In Figure 4.8 we display the performance of the different techniques when the size of the used data is 32KiB (i.e. the vector has 8192 elements). Note that the y-axis presents a logarithmic scale. The x-axis presents how many reader threads were active during the measurement. Similarly to Figure 4.8, Figure 4.9 and 4.10

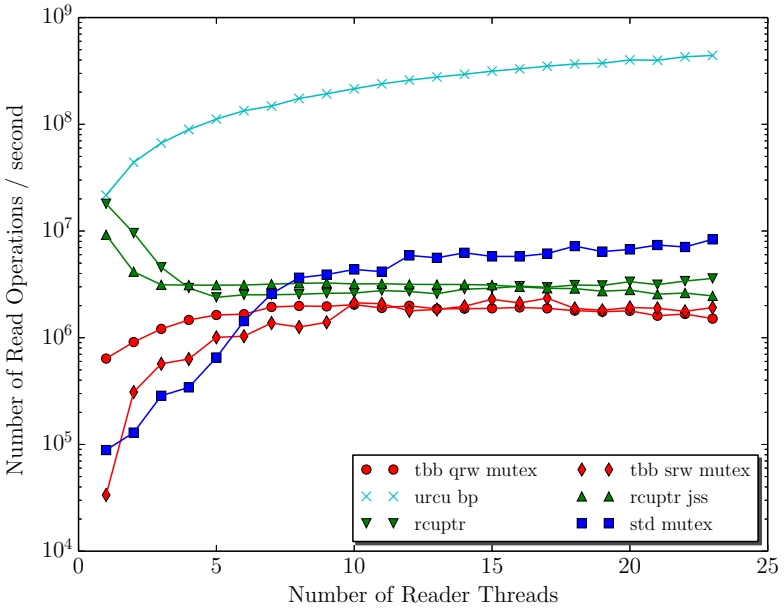


Figure 4.8: Read-side performance, data size: 32KiB

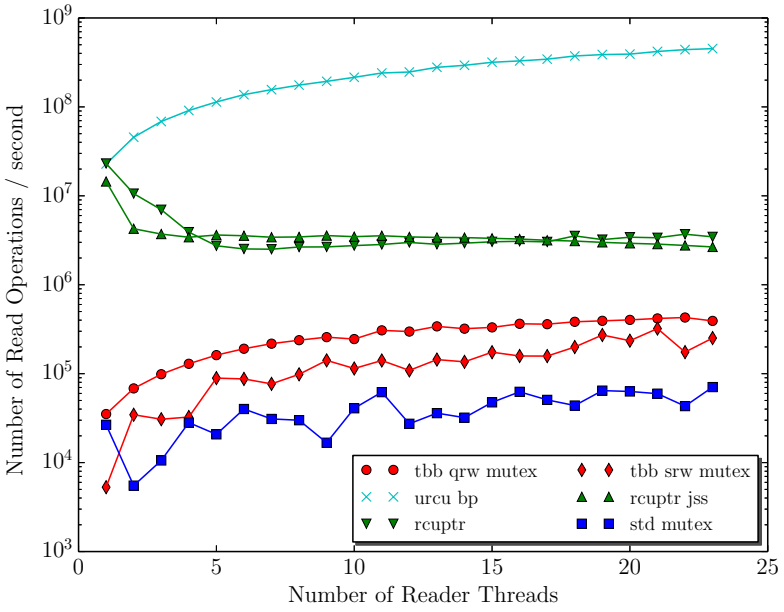


Figure 4.9: Read-side performance, data size: 512KiB

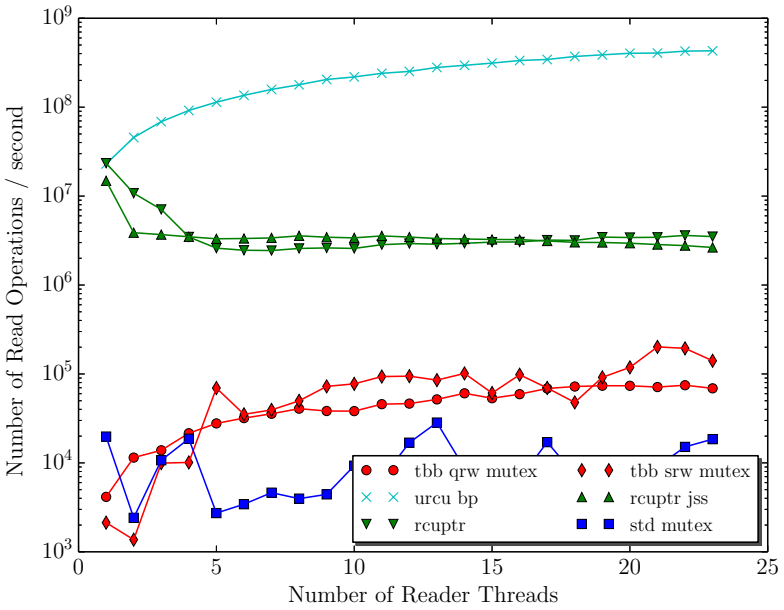


Figure 4.10: Read-side performance, data size: 4MiB

show the read performance in case of 512KiB and 4MiB data size respectively. Figure 4.9 and 4.10 illustrate that our `rcu_ptr` implementation can outperform the traditional mutex based implementation with more than two orders of magnitude. Also, `rcu_ptr` can outperform the read-write mutex based realizations with more than one order of magnitude. The `rcu_ptr` based techniques have some degradation until the readers' number is less than 5 (approximately). From that point, the performance has no or minimal degradation as depicted in Figure 4.11: The lock-free implementation of the atomic `shared_ptr` results in some minimal degradation, while the spin lock based one provides a slight increase in the performance. We suspect that the reason behind this minimal degradation in case of the lock-free `shared_ptr` version is caused by the saturation of the system by the double world CAS operations. This slight increase (or even a minimal degradation) is in contrast to the read-write mutex and the URCU based methods, where the performance is growing continuously and in a faster pace as the number of the readers grows.

Compared to URCU, our technique can be outperformed up to two orders of magnitudes. This is the price we pay for the higher level of abstraction and for the general usability: we lose most of the performance because of the extra administration done with the reference counting in the underlying `shared_ptr` implementations while the bulletproof URCU uses only memory barrier instruc-

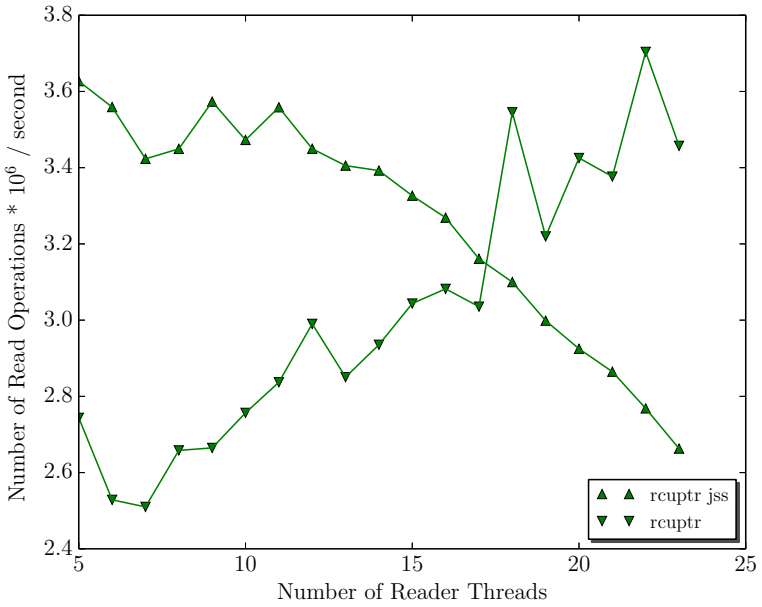


Figure 4.11: Read-side performance of variants of the rcu_ptr, data size: 512KiB

tions.

Figure 4.12 presents that RCU write-side performance is outperformed by the mutex variants (32KiB data size). This is the expected behaviour since RCU solutions are tuned for the read-side performance, but this implies some trade-offs on the write-side. However, our technique can outperform the bulletproof version of URCU in write-side performance. E.g, when the data size is 512 KiB then our method can be twice as fast (Figure 4.13). This is because with `urcu bp` one cannot use the `call_rcu()` to deallocate memory asynchronously, thus the writer thread must wait for all the pre-existing readers to be completed. This wait is done by `synchronize_rcu()` function and the duration actually waited is called an RCU grace period. Regarding write-side performance, we measured that all RCU based approaches are outperformed by all the mutex based solutions. The difference can be up to 20x, based on the used RCU and mutex implementation and on the size of the data. Interestingly, our measurements show that the lock-free implementation of `shared_ptr` does not provide higher read or write performance compared to the non lock-free version.

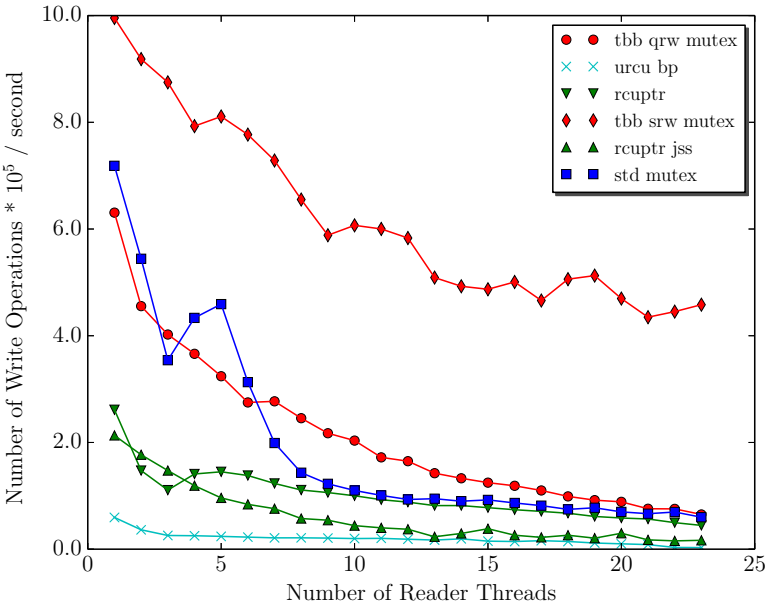


Figure 4.12: Write-side performance, data size: 32KiB

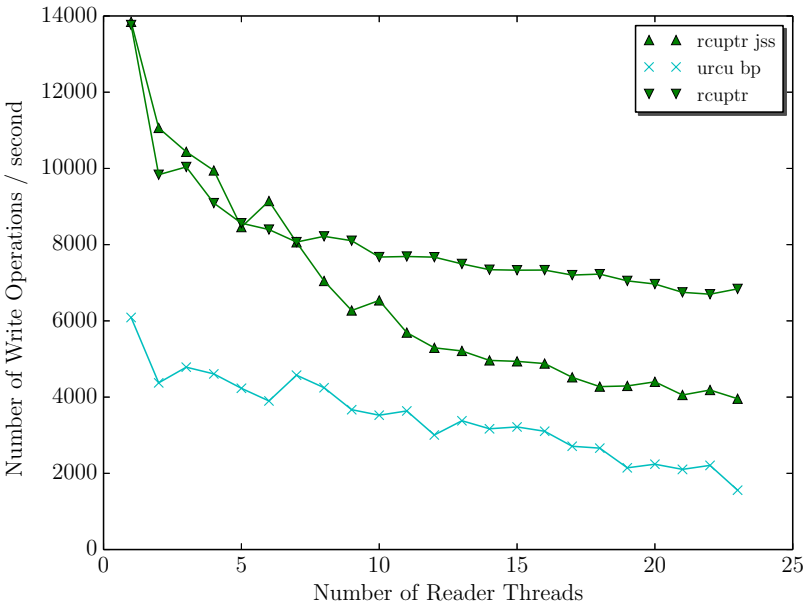


Figure 4.13: Write performance of RCU, data size: 512KiB

4.5 Correctness and Testing

To validate the correctness of our data structure we used different testing methods. We executed unit tests in a sequential manner (i.e. no parallel execution) to validate the basic behaviour of the class template. We used oriented stress testing [223] and sanitizers from the LLVM/Clang infrastructure [82] to verify behaviour during concurrent execution. During our stress tests, we focused on pairs of public methods of `rcu_ptr` and we executed these functions from different threads. We executed the operations in a loop on each thread and we added random delays in between each call. This way we tested different execution timings and we could make race windows slightly larger.

4.6 Future Work

It is our ongoing work to create performance measurements of our `rcu_ptr` on a weekly ordered architecture like ARMv7 as well. In order to reach the consume semantics in `rcu_ptr`, we may use hardware specific instructions in the future to overcome the problem of the deprecated `memory_order_consume`.

4.7 Conclusion

RCU is a technique in concurrent programming which is getting used more and more often nowadays. It has been introduced in the Linux kernel first, but the efficiency of the technique became proven so people demanded an implementation which could be used in user space too. The currently available user space RCU solutions do not provide a mechanism for automatic memory reclamation, also they provide a low-level C API, which may be prone to errors. In this section, we presented a high-level C++ implementation for the read-copy-update pattern, which provides automatic memory deallocation. Our technique complements the existing user space RCU implementation by providing a well performing safe and hard-to-misuse library. Thus, this library may be a good default choice by C++ developers who expect more readers than writers in their application.

4.8 Contribution

Thesis 4 (High-level abstraction for the read-copy-update pattern)**.** *I have presented a high-level abstraction to utilize the read-copy-update pattern and to support efficient concurrent programming in C++. I have designed a library to optimise performance in a heavily multithreaded environment, at the same time providing high-level abstractions, like smart pointers and other C++11/14/17 features. I have demonstrated that the technique complements the existing user space RCU implementation by providing a well performing, safe and hard-to-misuse library.*

thesis name	relevant publications									
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
(1) New non-intrusive testing methods	∘	∘	∘	∘					∘	
(2) Extending access for non-intrusive and white-box testing	∘							∘		∘
(3) Selective friend					∘					
(4) **High-level abstraction for the read-copy-update pattern**						•	•			

Note that the related publications have another author (Imre Szekeres[2]) besides my advisor and me.

[2]Imre created the wrapper above the free atomic function overloads of `std::shared_ptr` and we discussed C++ memory order related issues. His overall contribution is roughly 20-30% of the whole research. I have created the `rcu_ptr` smart pointer library including the implementation, the public API and the memory order related details. Also, I have executed the measurements with the different variants of `rcu_ptr` and with the various existing synchronization mechanisms and evaluated the results.

Chapter 5

Summary

This dissertation presented novel research results in three fundamental areas of software development: testing, encapsulation and abstraction. The first two theses are centred around non-intrusive testing, a testing technique which does not require any structural modification in the production code. In Chapter 2 we discussed the existing non-intrusive testing methods and we enlisted their advantages and disadvantages. We introduced our new method which is based on function call interception and has numerous clear benefits compared to preexisting efforts. With this method, we can replace functions with test doubles even if they are inline functions. Furthermore, we described two new experimental approaches which make it possible to substitute types with test double types: one which exploits syntax tree transformations and another which is based on compile-time reflection (Thesis 1).

We presented that often it is needed to access private members in order to have non-intrusive tests. To solve this problem, we introduced new techniques to access private members of a class for the purpose of non-intrusive or white-box testing: a library based on explicit template instantiation and out-of-class friends (Thesis 2).

Regarding encapsulation, we demonstrated that certain language constructs like the `friend` in C++ may provide exaggerated access to the internals of a class. This excessive access may be the source of errors in the software. We suggested a new language construct which makes it possible to restrict access of a friend only to a certain well-specified set of members, this way it strengthens encapsulation and information hiding (Chapter 3, Thesis 3).

Besides encapsulation, abstraction plays an essential role in large scale software system development, especially when multiple threads of execution are involved. We demonstrated a new high-level abstraction for the read-copy-update concurrency pattern, which provides reasonable performance meanwhile it gives a generic and safe to use C++ API (Chapter 4, Thesis 4).

5.1 Results

Thesis 1 (New non-intrusive testing methods)**.** *I have analysed the existing dependency replacement techniques of C++ for testing and evaluated their advantages and disadvantages. I have introduced and analysed three new non-intrusive testing approaches: (1) I have implemented a method based on compiler instrumentation and function call interception. The new technique has clear advantages, thus it provides an alternative way to replace dependencies. I have created and evaluated a prototype implementation which is publicly available. (2) I have presented an experimental procedure which transforms the original abstract syntax tree of the production code for testing. With this procedure, it is possible to replace not just simple functions but also types. I have created a proof-of-concept prototype to demonstrate that the idea is feasible. (3) I have proposed a static reflection based approach as a future direction. Besides replacing types this solution could be used to implement generic proxy and mock objects for unit test frameworks.*

Thesis 2 (Extending access for non-intrusive and white-box testing)**.** *I have analysed the various existing methods available for accessing private members in C++. To support non-intrusive and white-box testing I have developed two different approaches eliminating the existing drawbacks. (1) I have created a library which exploits the explicit template instantiation mechanism of C++ and this way enables access to private members. Currently, this library is the only generic solution to access private members without violating the C++ standard. (2) I have presented how friend declarations added outside of a class could provide a full, non-intrusive solution to separate test related code from the source of the unit under test. I have realized a prototype based on C++ attributes to justify the feasibility of out-of-class friends.*

Thesis 3 (Selective friend)**.** *I have investigated how the friendship mechanism is used in C++ programs. I have shown that there are various holes and errors in friend usage like friend functions accessing only public members or not accessing members at all. I have proposed a selective friend language construct for C++ which can restrict friendship only to well-defined members. I have demonstrated that such a new language element may decrease the degradation of encapsulation and significantly increase the diagnostic capacity of the compiler. To underpin my statements, I have made a publicly available friend inspection tool, which can list the possible erroneous uses of friend declarations and I have created a proof-of-concept implementation of selective friends based on the LLVM/Clang compiler infrastructure.*

Thesis 4 (High-level abstraction for the read-copy-update pattern)**.** *I have presented a high-level abstraction to utilize the read-copy-update pattern and to support*

efficient concurrent programming in C++. I have designed a library to optimise performance in a heavily multithreaded environment, at the same time providing high-level abstractions, like smart pointers and other C++11/14/17 features. I have demonstrated that the technique complements the existing user space RCU implementation by providing a well performing, safe and hard-to-misuse library.

thesis name	relevant publications									
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
(1) New non-intrusive testing methods	•	•	•	•					•	
(2) Extending access for non-intrusive and white-box testing	•							•		•
(3) Selective friend					•					
(4) High-level abstraction for the read-copy-update pattern						•	•			

References

[1] G. Márton and Z. Porkoláb, "Unit Testing in C++ with Compiler Instrumentation and Friends," *ACTA CYBERNETICA*, vol. 23, no. 2, pp. 659–686, 2017.

[2] G. Márton and Z. Porkoláb, "Compile-Time Function Call Interception for Testing in C/C++," *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA*, vol. 63, no. 1, pp. 17–32, 2018.

[3] G. Márton and Z. Porkoláb, *Utilize Syntax Tree Transformations as a C/C++ Test Seam*. Novi Sad; Beograd: CEUR-WS.org, 2018, ch. 10, pp. 1–8.

[4] G. Márton and Z. Porkoláb, "C++ Compile-time Reflection and Mock Objects," *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA*, vol. LIX, pp. 5–20, 2014.

[5] G. Márton and Z. Porkoláb, "Selective friends in C++," *SOFTWARE-PRACTICE & EXPERIENCE*, vol. 48, no. 8, pp. 1493–1519, 2018.

[6] G. Márton, I. Szekeres, and Z. Porkoláb, "Towards a High-level C++ Abstraction to Utilize the Read-Copy-Update Pattern," *ACTA ELECTROTECHNICA ET INFORMATICA*, vol. 18, no. 3, pp. 03–10, 2018.

[7] G. Márton, I. Szekeres, and Z. Porkoláb, *High Level C++ Implementation of the Read-Copy-Update Pattern*. Košice: IEEE Hungary Section, 2017, pp. 243–348.

[8] G. Márton and Z. Porkoláb, "Unit Testing and Friends in C++," Marosvásárhely, 2015.09.04 2015.09.02.

[9] G. Márton and Z. Porkoláb, "Compile-Time Function Call Interception to Mock Functions in C/C++," https://www.youtube.com/watch?v=mv60fYkKNHc, Bristol, England, April 2018, presentation at EuroLLVM Conference.

[10] G. Márton and Z. Porkoláb, "Friendship in Service of Testing," https://www.youtube.com/watch?v=U9Up_OfiW24, Aspen, USA, May 2016, presentation at C++Now Conference.

[11] G. Márton and Z. Porkoláb, "Unit testing and friends in c++," R. Ferenc, B. Bánhelyi, T. Gergely, A. Kertész, and Z. Kincses, Eds. Szeged: University of Szeged, Institute of Informatics, 2016.06.29 2016.06.27, p. 40.

[12] G. Márton and Z. Porkoláb, "Journey to C++ Compile-time Reflection," http://www.meetup.com/Hungarian-Cpp-Community/events/213166252/, Budapest, Hungary, October 2014, presentation at Hungarian C++ Community Meetup.

[13] G. Márton and Z. Porkoláb, "Friendship in Service of Testing," http://www.meetup.com/Hungarian-Cpp-Community/events/227145136/, Budapest, Hungary, December 2015, presentation at Hungarian C++ Community Meetup.

[14] G. Márton. (2018) Instrumentation for testing. [Online]. Available: https://github.com/martong/finstrument_mock

[15] G. Márton. (2018) Modified Clang ASTImporter for testing. [Online]. Available: https://github.com/martong/clang/tree/mock_with_astimporter

[16] G. Márton. (2014) Extended Clang reflection prototype with __record_member_function_count. [Online]. Available: https://github.com/martong/clang-reflection/tree/member_function

[17] G. Márton. (2016) Access private. [Online]. Available: https://github.com/martong/access_private

[18] G. Márton. (2016) Out-of-class friend. [Online]. Available: https://github.com/martong/clang/tree/out-of-class_friend_attr

[19] G. Márton. (2015) Friend statistics. [Online]. Available: https://github.com/martong/friend-stats

[20] G. Márton. (2017) Selective friend. [Online]. Available: https://github.com/martong/clang/tree/selective_friend

[21] G. Márton. (2018) rcu_ptr. [Online]. Available: https://github.com/martong/rcu_ptr

[22] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing.* John Wiley & Sons, 2011.

[23] D. Spinellis, "State-of-the-art software testing," *IEEE Software*, vol. 34, no. 5, pp. 4–6, 2017.

[24] S. Planning, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology*, 2002.

[25] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[26] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[27] D. Janzen and H. Saiedian, "Test-driven development concepts, taxonomy, and future direction," *Computer*, vol. 38, no. 9, pp. 43–50, 2005.

[28] H. Sutter and A. Alexandrescu, *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[29] M. Siniaalto and P. Abrahamsson, "Does test-driven development improve the program code? alarming results from a comparative case study," in *Balancing Agility and Formalism in Software Engineering*. Springer, 2008, pp. 143–156.

[30] M. Feathers, *Working Effectively with Legacy Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[31] O. Nierstrasz, "A survey of object-oriented concepts," in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. H. Lochovsky, Eds. New York: ACM PRESS, 1989, pp. 3–21.

[32] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages." *SIGPLAN Not.*, vol. 21, no. 11, pp. 38–45, 1986.

[33] N. Schärli, A. P. Black, and S. Ducasse, "Object-oriented encapsulation for dynamically typed languages," in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '04. New York, NY, USA: ACM, 2004, pp. 130–149. [Online]. Available: http://doi.acm.org/10.1145/1028976.1028988

[34] isocpp.org. (2016) C++ FAQ, Friends, Do friends violate encapsulation? Https://isocpp.org/wiki/faq/friends#friends-and-encap. [Online]. Available: https://isocpp.org/wiki/faq/friends#friends-and-encap

[35] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *IEEE transactions on software engineering*, vol. 21, no. 4, pp. 314–335, 1995.

[36] J. Greenfield and K. Short, "Software factories: assembling applications with patterns, models, frameworks and tools," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2003, pp. 16–27.

[37] ISO, *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2017.

[38] B. Stroustrup, "Exception safety: concepts and techniques," in *Advances in exception handling techniques*. Springer, 2001, pp. 60–76.

[39] P. E. McKenney, *Is Parallel Programming Hard, And, If So, What Can You Do About It?* Corvallis, OR, USA: kernel.org, 2010. [Online]. Available: http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html

[40] A. Bertolino and E. Marchetti, "1 a brief essay on software testing," 2004.

[41] M. E. Khan and F. Khan, "Importance of software testing in software development life cycle," 2014.

[42] M. Fowler. The new methodology. Http://www.martinfowler.com/articles/newMethodology.html. [Online]. Available: http://www.martinfowler.com/articles/newMethodology.html

[43] T. A. Majchrzak, *Improving Software Testing: Technical and Organizational Developments*. Springer Publishing Company, Incorporated, 2012.

[44] T. Winters and H. Wright, "All your tests are terrible," 2015.

[45] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[46] wikipedia.org. Service locator pattern. Https://goo.gl/1KFxKj https://en.wikipedia.org/wiki/Service_locator_pattern. [Online]. Available: https://en.wikipedia.org/wiki/Service_locator_pattern

[47] M. Fowler. Using a service locator. Http://martinfowler.com/articles/injection.html#UsingAServiceLocator. [Online]. Available: http://martinfowler.com/articles/injection.html#UsingAServiceLocator

[48] N. Schwarz, M. Lungu, and O. Nierstrasz, "Seuss: Decoupling responsibilities from static methods for fine-grained configurability," *Journal of Object Technology*, vol. 11, no. 1, pp. 3:1–23, Apr. 2012. [Online]. Available: http://www.jot.fm/contents/issue_2012_04/article3.html

[49] M. Seemann, *Dependency Injection in .NET*, M. Seemann, Ed. Manning, 2011.

[50] wikipedia.org. Dependency injection. Https://goo.gl/OjlpOY http://en.wikipedia.org/wiki/Dependency_injection. [Online]. Available: http://en.wikipedia.org/wiki/Dependency_injection

[51] M. D. Network. Unity container. Https://goo.gl/YQwWlE https://msdn.microsoft.com/en-us/library/ff647202.aspx. [Online]. Available: https://msdn.microsoft.com/en-us/library/ff647202.aspx

[52] D. Mane, N. Ojha, and K. Chitnis, "The spring framework: An open source java platform for developing robust java applications," *International Journal of Innovative Technology and Exploring Engineering*. [Online]. Available: http://www.ijitee.org/attachments/File/v3i2/B1010073213.pdf

[53] M. Rüegg and P. Sommerlad, "Refactoring towards seams in c++," in *Proceedings of the 7th International Workshop on Automation of Software Test*, ser. AST '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 117–123. [Online]. Available: http://dl.acm.org/citation.cfm?id=2663608.2663632

[54] mockator.com. An eclipse cdt plug-in for c++ seams and mock objects. Http://mockator.com/. [Online]. Available: http://mockator.com/

[55] J. Mihalicza, Z. Porkoláb, and A. Gabor, "Type-preserving heap profiler for C++," in *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*. IEEE Computer Society, 2011, pp. 457–466. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2011.6080813

[56] gcc.gnu.org. (2019) The c preprocessor: Search path. [Online]. Available: https://gcc.gnu.org/onlinedocs/cpp/Search-Path.html

[57] B. Stroustrup, H. Sutter *et al.* (2016) C++ core guidelines. Https://goo.gl/7ZXiov https://github.com/isocpp/-/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rr-scoped. [Online]. Available: https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rr-scoped

[58] K. Driesen and U. Holzle, "The Direct Cost of Virtual Function Calls in C++," in *Proceedings of the 11th Conference on Object-Oriented Programming Systems, Languages and Applications*, L. Anderson and J. Coplien, Eds. ACM, 1996, pp. 306–323.

[59] S. Parent, "Inheritance is the base class of evil," 2013.

[60] S. Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005, item 41, Understand implicit interfaces and compile-time polymorphism.

[61] W. Bright. (2010, August) C++ compilation speed. Http://www.drdobbs.com/cpp/c-compilation-speed/228701711. [Online]. Available: http://www.drdobbs.com/cpp/c-compilation-speed/228701711

[62] S. Meyers, *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*, 1st ed. O'Reilly Media, Inc., 2014.

[63] ISO, *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2014.

[64] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[65] S. Venkatakrishnan. Build operate check clear - test pattern. [Online]. Available: http://developer-in-test.blogspot.hu/2009/05/build-operate-check-clear-test-pattern.html

[66] D. North, "Introducing bdd, better software magazine," 2006.

[67] M. Fowler. Givenwhenthen
. [Online]. Available: https://martinfowler.com/bliki/GivenWhenThen.html

[68] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[69] P. Kang, "Function call interception techniques," *Software: Practice and Experience*, pp. n/a–n/a, spe.2501. [Online]. Available: http://dx.doi.org/10.1002/spe.2501

[70] L. P. Manual. (2017) ld.so, ld-linux.so - dynamic linker/loader. [Online]. Available: http://man7.org/linux/man-pages/man8/ld.so.8.html

[71] L. P. Manual. (2017) dlsym, dlvsym - obtain address of a symbol in a shared object or executable. [Online]. Available: http://man7.org/linux/man-pages/man3/dlsym.3.html

[72] Intel, CodeSourcery, Compaq, EDG, HP, IBM, R. Hat, and SGI. (2017) Itanium c++ abi. [Online]. Available: http://refspecs.linuxbase.org/cxxabi-1.83.html

[73] L. P. Manual. (2017) ptrace - process trace. [Online]. Available: http://man7.org/linux/man-pages/man2/ptrace.2.html

[74] P. Padala, "Playing with ptrace, Part I," vol. 103, Nov. 2002.

[75] gnu.org. (2017) Gdb: The gnu project debugger. [Online]. Available: https://www.gnu.org/software/gdb/

[76] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[77] Intel. (2017) Pintool api reference - rtn: Routine object. [Online]. Available: https://software.intel.com/sites/landingpage/pintool/docs/53271/Pin/html/group___RTN___BASIC___API.html

[78] gnu.org. (2017) Using gnu ld. [Online]. Available: ftp://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html

[79] K. Avijit, P. Gupta, and D. Gupta, "Binary rewriting and call interception for efficient runtime protection against buffer overflows: Research articles," *Softw. Pract. Exper.*, vol. 36, no. 9, pp. 971–998, Jul. 2006. [Online]. Available: http://dx.doi.org/10.1002/spe.v36:9

[80] gcc.gnu.org. (2017) Program instrumentation options. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

[81] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[82] llvm.org. (2016) clang: a c language family frontend for llvm. Http://clang.llvm.org. [Online]. Available: http://clang.llvm.org

[83] llvm.org. (2017) Llvm language reference manual. [Online]. Available: http://llvm.org/docs/LangRef.html

[84] llvm.org. (2019) Clang cfe internals manual. [Online]. Available: https://clang.llvm.org/docs/InternalsManual.html

[85] L. P. Manual. (2017) mmap, munmap - map or unmap files or devices into memory. [Online]. Available: http://man7.org/linux/man-pages/man2/mmap.2.html

[86] gcc.gnu.org. (2017) Declaring attributes of functions. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-4.3.0/gcc/Function-Attributes.html

[87] gcc.gnu.org. (2017) Extracting the function pointer from a bound pointer to member function. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-4.9.0/gcc/Bound-member-functions.html

[88] llvm.org. (2017) Clang will not accept a conversion from a bound pmf to a regular method pointer. [Online]. Available: https://bugs.llvm.org/show_bug.cgi?id=22121

[89] Adobe. (2017) C++ performance benchmarks. [Online]. Available: https://stlab.adobe.com/performance

[90] M. Müller, "Abstraction benchmarks and performance of c++ applications," in *Proceedings of the Fourth International Conference on Supercomputing in Nuclear Applications*. Citeseer, 2000.

[91] K. Chen, S. Chan, R.-C. Ju, and P. Tu, "Optimizing structures in object oriented programs," in *9th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT'05)*. IEEE, 2005, pp. 94–103.

[92] A. Stepanov. Stepanov benchmark. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/D_3.cpp

[93] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1273442.1250746

[94] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 213–223. [Online]. Available: http://dl.acm.org/citation.cfm?id=2190025.2190067

[95] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *In Proc. of the Winter 1992 USENIX Conference*, 1991, pp. 125–138.

[96] Intel. (2017) Intel inspector. [Online]. Available: https://software.intel.com/en-us/intel-inspector-xe

[97] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 28–28. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342849

[98] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09. New York, NY, USA: ACM, 2009, pp. 62–71. [Online]. Available: http://doi.acm.org/10.1145/1791194.1791203

[99] D. M. Berris, A. Veitch, N. Heintze, E. Anderson, and N. Wang, "Xray: A function call tracing system," 2016.

[100] llvm.org. (2017) Xray instrumentation. [Online]. Available: https://llvm.org/docs/XRay.html

[101] llvm.org. (2017) Memory sanitizer. [Online]. Available: https://clang.llvm.org/docs/MemorySanitizer.html

[102] llvm.org. (2017) Undefined behavior sanitizer. [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[103] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proceedings of the 3rd International Conference on Virtual Execution Environments*, ser. VEE '07. New York, NY, USA: ACM, 2007, pp. 65–74. [Online]. Available: http://doi.acm.org/10.1145/1254810.1254820

[104] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "Tainttrace: Efficient flow tracing with dynamic binary rewriting," in *Proceedings of the 11th IEEE Symposium on Computers and Communications*, ser. ISCC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 749–754. [Online]. Available: http://dx.doi.org/10.1109/ISCC.2006.158

[105] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *Proceedings of the 39th Annual IEEE/ACM International*

Symposium on Microarchitecture, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 135–148. [Online]. Available: https://doi.org/10.1109/MICRO.2006.29

[106] M. Brünink, M. Süßkraut, and C. Fetzer, "Boundless memory allocations for memory safety and high availability," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, June 2011, pp. 13–24.

[107] Q. Zhao, D. Bruening, and S. Amarasinghe, "Efficient memory shadowing for 64-bit architectures," in *Proceedings of the 2010 International Symposium on Memory Management*, ser. ISMM '10. New York, NY, USA: ACM, 2010, pp. 93–102. [Online]. Available: http://doi.acm.org/10.1145/1806651.1806667

[108] Q. Zhao, D. Bruening, and S. Amarasinghe, "Umbra: Efficient and scalable memory shadowing," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 22–31. [Online]. Available: http://doi.acm.org/10.1145/1772954.1772960

[109] N. Hasabnis, A. Misra, and R. Sekar, "Light-weight bounds checking," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 135–144. [Online]. Available: http://doi.acm.org/10.1145/2259016.2259034

[110] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison Wesley*, vol. 7, no. 8, p. 9, 1986.

[111] D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet, *Groovy in action*. Manning Publications Co., 2007.

[112] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD Conference*, 2008, pp. 1–2.

[113] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser, "Design of the codeboost transformation system for domain-specific optimisation of c++ programs," in *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*. IEEE, 2003, pp. 65–74.

[114] O. S. Bagge and M. Haveraaen, "Domain-specific optimisation with user-defined rules in CodeBoost," in *Proceedings of the 4th International Workshop on Rule-Based Programming (RULE'03)*, ser. Electronic Notes in Theoretical Computer Science, J.-L. Giavitto and P.-E. Moreau, Eds., vol. 86/2. Valencia, Spain: Elsevier, 2003.

[115] K. Olmos and E. Visser, "Strategies for source-to-source constant propagation," in *Workshop on Reduction Strategies (WRS'02)*, ser. Electronic Notes in Theoretical Computer Science, B. Gramlich and S. Lucas, Eds., vol. 70, no. 6. Copenhagen, Denmark: Elsevier Science Publishers, July 2002, p. 20, http://www.elsevier.nl/locate/entcs/volume70.html.

[116] B. Fischer and E. Visser, "Adding concrete syntax to a prolog-based program synthesis system," in *International Symposium on Logic-Based Program Synthesis and Transformation*. Springer, 2003, pp. 56–58.

[117] G. Horvath, "Cross translation unit analysis in clang static analyzer," 2017, euroLLVM. [Online]. Available: https://www.youtube.com/watch?v=7AWgaqvFsgs

[118] Oracle. (2015) Trail: The reflection api. [Online]. Available: http://docs.oracle.com/javase/tutorial/reflect/

[119] M. D. Network. (2015) Reflection (c# and visual basic). [Online]. Available: http://msdn.microsoft.com/en-us/library/ms173183.aspx

[120] (2015) Scala documentation, reflection. [Online]. Available: http://docs.scala-lang.org/overviews/reflection/overview.html

[121] (2015) D programming language, traits. [Online]. Available: http://dlang.org/traits.html

[122] R. Ramey. (2004) Boost serialization library. [Online]. Available: http://www.boost.org/doc/libs/1_55_0/libs/serialization/doc/index.html

[123] Google. (2017) Google test. [Online]. Available: https://github.com/google/googletest

[124] J. de Guzman, D. Marsden, and T. Schwinger. (2001) Boost fusion library. [Online]. Available: http://www.boost.org/doc/libs/1_55_0/libs/fusion/doc/html/

[125] J. Snyder and C. Carruth. (2013) Call for compile-time reflection proposals (n3814). [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3814.html

[126] A. Tomazos and C. Käser. (2013) Enumerator list property queries (n3815). [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3815.html

[127] A. Tomazos and C. Käser. (2014) Type member property queries. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4027.pdf

[128] A. Tomazos and C. Käser. (2014) Reflection type traits for classes, unions and enumerations. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4113.pdf

[129] A. Tomazos and C. Käser. (2015) Type property queries. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4428.pdf

[130] C. Kaeser. (2013) Clang reflection - n3815 reference implementation. [Online]. Available: https://github.com/ChristianKaeser/clang-reflection

[131] C. S. Silva and D. Auresco. (2014) C++ type reflection via variadic template expansion (n3951). [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3951.pdf

[132] C. S. Silva and D. Auresco. (2016) C++ static reflection via template pack expansion (p0255r0). [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0255r0.pdf

[133] M. Chochlik, A. Naumann, and D. Sankel. (2016) Static reflection rationale, design and evolution (p0385r1). [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0385r1.pdf

[134] P. Németh. (2014) Code checkers & generators (n3883). [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3883.html

[135] A. Sutton and H. Sutter. (2017) A design for static reflection (p0590r0). [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0590r0.pdf

[136] M. Price. (2014) Compile-time string draft (d3933). [Online]. Available: https://groups.google.com/a/isocpp.org/forum/#!forum/reflection

[137] S. Chiba, "A metaobject protocol for c++," in *ACM Sigplan Notices*, vol. 30, no. 10. ACM, 1995, pp. 285–299.

[138] S. Chiba, "Open c++ programmer's guide for version 2," SPL-96-024, Xerox PARC, Tech. Rep., 1996.

[139] Oracle. (2014) Java platform standard ed. 7, class proxy. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html

[140] llvm.org. (2018) Clang cfe internals manual - how to add an expression or statement. [Online]. Available: https://clang.llvm.org/docs/InternalsManual.html#how-to-add-an-expression-or-statement

[141] G. D. Reis. A module system for c++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4047.pdf. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4047.pdf

[142] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil *et al.*, "Analyzing parallel programs with pin," *Computer*, vol. 43, no. 3, pp. 34–41, 2010.

[143] S. Nidhra and J. Dondeti, "Black box and white box testing techniques-a literature review," *International Journal of Embedded Systems and Applications (IJESA)*, vol. 2, no. 2, pp. 29–50, 2012.

[144] M. E. Khan, F. Khan *et al.*, "A comparative study of white box, black box and grey box testing techniques," *Int. J. Adv. Comput. Sci. Appl*, vol. 3, no. 6, 2012.

[145] Oracle. Java platform, standard edition 6 api specification, class accessibleobject. Https://goo.gl/rfKFRA https://docs.oracle.com/javase/6/docs/-/api/java/lang/reflect/AccessibleObject.html. [Online]. Available: https://docs.oracle.com/javase/6/docs/api/java/lang/reflect/AccessibleObject.html

[146] llvm.org. Llvm programmer's manual. http://releases.llvm.org/3.6.0/docs/ProgrammersManual.html. [Online]. Available: http://releases.llvm.org/3.6.0/docs/ProgrammersManual.html

[147] J. Schaub. Access to private members: Safer nastiness. Https://goo.gl/aG6HEv http://bloglitb.blogspot.hu/2010/07/access-to-private-members-thats-easy.html. [Online]. Available: http://bloglitb.blogspot.hu/2010/07/access-to-private-members-thats-easy.html

[148] C. Kumar, *Advanced C++ Faqs: Volumes 1 & 2.* Createspace Independent Pub, 2014. [Online]. Available: https://books.google.hu/books?id=pMTroAEACAAJ

[149] N. Schärli, "Composable encapsulation policies," ser. ECOOP '04, 2004, pp. 248–274.

[150] J. Hogg, "Islands: Aliasing protection in object-oriented languages," in *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '91. New York, NY, USA: ACM, 1991, pp. 271–285. [Online]. Available: http://doi.acm.org/10.1145/117954.117975

[151] P. S. Almeida, "Balloon types: Controlling sharing of state in data types," in *Proceedings ECOOP'97*, ser. LNCS, vol. 1241. Springer-Verlag, Jun. 1997, pp. 32–59.

[152] J. Noble, J. Vitek, and J. Potter, "Flexible alias protection," in *Proceedings ECOOP'98*, ser. LNCS, 1998, pp. 158–185.

[153] J. Aldrich, V. Kostadinov, and C. Chambers, "Alias annotations for program understanding," *SIGPLAN Not.*, vol. 37, no. 11, pp. 311–330, Nov. 2002. [Online]. Available: http://doi.acm.org/10.1145/583854.582448

[154] G. Kniesel and D. Theisen, "JAC - Java with transitive readonly access control," in *Proceedings of the Intercontinental Workshop on Aliasing in Object-Oriented Systems*, Lisbon, Portugal, Jun. 14-18 1999.

[155] C. Boyapati, B. Liskov, and L. Shrira, "Ownership types for object encapsulation," *SIGPLAN Not.*, vol. 38, no. 1, pp. 213–223, Jan. 2003. [Online]. Available: http://doi.acm.org/10.1145/640128.604156

[156] D. Eiffel. (2016) Adding class features, making features available to clients. [Online]. Available: https://docs.eiffel.com/book/platform-specifics/adding-class-features#Making_Features_Available_to_Clients

[157] isocpp.org. (2016) C++ FAQ, Friends, Should my class declare a member function or a friend function? Https://isocpp.org/wiki/faq/friends#member-vs-friend-fns. [Online]. Available: https://isocpp.org/wiki/faq/friends#member-vs-friend-fns

[158] B. Stroustrup, *The C++ programming language*. Pearson Education, 2013.

[159] B. Stroustrup, *The Design and Evolution of C++*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1994, p. 53.

[160] A. R. Bolton. (2006, January) Friendship and the attorney-client idiom. Http://www.drdobbs.com/friendship-and-the-attorney-client-idiom/184402053. [Online]. Available: http://www.drdobbs.com/friendship-and-the-attorney-client-idiom/184402053

[161] T. Misfeldt, G. Bumgardner, A. Gray, and L. Xiaoping, *The Elements of C++ Style*. Cambridge University Press, 2004, p. 77.

[162] N. Josuttis, *Object-Oriented Programming in C++*. Wiley, 2002.

[163] Oracle. (2015) Controlling access to members of a class. Https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

[164] Stackoverflow. (2008) Is there a way to simulate the c++ 'friend' concept in java? Http://stackoverflow.com/questions/182278/is-there-a-way-to-simulate-the-c-friend-concept-in-java. [Online]. Available: http://stackoverflow.com/questions/182278/is-there-a-way-to-simulate-the-c-friend-concept-in-java

[165] Stackoverflow. (2009) Why can outer java classes access inner class private members? Http://stackoverflow.com/questions/1801718/why-can-outer-java-classes-access-inner-class-private-members. [Online]. Available: http://stackoverflow.com/questions/1801718/why-can-outer-java-classes-access-inner-class-private-members

[166] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. (2016) The java language specification, java se 7 edition, determining accessibility. [Online]. Available: http://docs.oracle.com/javase/specs/jls/se7/html/jls-6.html#jls-6.6.1

[167] D. N. Microsoft. (2016) Access modifiers (c# reference). [Online]. Available: https://msdn.microsoft.com/en-us/library/wxh6fsc7.aspx

[168] Stackoverflow. (2008) What is the c# equivalent of friend? [Online]. Available: http://stackoverflow.com/questions/204739/what-is-the-c-sharp-equivalent-of-friend

[169] D. N. Microsoft. (2016) Internalsvisibletoattribute class. [Online]. Available: https://msdn.microsoft.com/en-us/library/system.runtime.compilerservices.internalsvisibletoattribute.aspx

[170] dlang.org. (2016) D programming language, friends. [Online]. Available: http://dlang.org/cpptod.html#friends

[171] N. D. Matsakis and F. S. Klock II, "The rust language," in *ACM SIGAda Ada Letters*, vol. 34, no. 3. ACM, 2014, pp. 103–104.

[172] The Rust Project Developers. (2018) Rust documentation. Accessed: 2018-02-20. [Online]. Available: https://doc.rust-lang.org/

[173] S. F. Python. (2016) The python language reference. [Online]. Available: https://docs.python.org/2/reference/

[174] Comprehensive Perl Archive Network. (2018) perltoot - tom's object-oriented tutorial for perl. Accessed: 2018-02-20. [Online]. Available: http://search.cpan.org/dist/perl-5.8.9/pod/perltoot.pod

[175] V. Savikko, "Design patterns in python," in *Proceedings of the 6th International Python Conference*. Citeseer, 1997.

[176] llvm.org. (2016) Clang 3.7 documentation, tutorial for building tools using libtooling and libastmatchers. Http://clang.llvm.org/docs/LibASTMatchersTutorial.html. [Online]. Available: http://clang.llvm.org/docs/LibASTMatchersTutorial.html

[177] boost.org. (2008) Boost operators library. [Online]. Available: http://www.boost.org/doc/libs/1_56_0/libs/utility/operators.htm

[178] J. O. Coplien, "Curiously recurring template patterns," *C++ Rep.*, vol. 7, no. 2, pp. 24–27, Feb. 1995. [Online]. Available: http://dl.acm.org/citation.cfm?id=229227.229229

[179] B. Schling, *The Boost C++ Libraries*. XML Press, 2011. [Online]. Available: https://theboostcpplibraries.com/

[180] boost.org. (2016) Boost c++ libraries. [Online]. Available: http://www.boost.org/

[181] o. t. N. I. o. H. US National Library of Medicine. (2016) Insight segmentation and registration toolkit (itk). [Online]. Available: http://www.itk.org/

[182] Qt. (2018) Cross-platform software development for embedded & desktop. Accessed: 2018-01-30. [Online]. Available: https://www.qt.io/

[183] ISO, *ISO/IEC 25436:2006 - Eiffel: Analysis, Design and Programming Language*. Geneva, Switzerland: International Organization for Standardization, 2006.

[184] wikpedia.org. (2016) Eiffel (programming language). [Online]. Available: http://en.wikipedia.org/wiki/Eiffel_(programming_language)

[185] M. English, J. Buckley, and T. Cahill, "A friend in need is a friend indeed [software metrics and friend functions]," in *Empirical Software Engineering, 2005. 2005 International Symposium on.* IEEE, 2005, pp. 10–pp.

[186] F. G. Wilkie and B. Kitchenham, "An investigation of coupling, reuse and maintenance in a commercial c++ application," *Information and Software Technology*, vol. 43, no. 13, pp. 801–812, 2001.

[187] S. Counsell and P. Newson, "Use of friends in c++ software: an empirical investigation," *Journal of Systems and Software*, vol. 53, no. 1, pp. 15–21, 2000.

[188] M. English, J. Buckley, T. Cahill, and T. Lynch, "An empirical study of the use of friends in c++ software," in *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on.* IEEE, 2005, pp. 329–332.

[189] M. English, J. Buckley, T. Cahill, K. Lynch, and M. English, "An analysis of the use of friends in c++ software systems."

[190] M. English, J. Buckley, and T. Cahill, "A replicated and refined empirical study of the use of friends in c++ software," *Journal of Systems and Software*, vol. 83, no. 11, pp. 2275–2286, 2010.

[191] M. English, T. Cahill, and J. Buckley, "Construct specific coupling measurement for c++ software," *Computer Languages, Systems & Structures*, vol. 38, no. 4, pp. 300–319, 2012.

[192] Stackoverflow. (2010) How to name this key-oriented access-protection pattern? [Online]. Available: http://stackoverflow.com/questions/3324248/how-to-name-this-key-oriented-access-protection-pattern

[193] A. Bergé. (2013) Tales of c++, friends with benefits. [Online]. Available: http://talesofcpp.fusionfenix.com/post-4/episode-three-friends-with-benefits

[194] wikibooks.org. (2014) More c++ idioms, friendship and the attorney-client. [Online]. Available: http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Friendship_and_the_Attorney-Client

[195] P. E. McKenney and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in *Parallel and Distributed Computing and Systems*, 1998, pp. 509–518.

[196] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni, "Read-copy update," in *AUUG Conference Proceedings*. AUUG, Inc., 2001, p. 175.

[197] J. M. Mellor-Crummey and M. L. Scott, "Scalable reader-writer synchronization for shared-memory multiprocessors," *SIGPLAN Not.*, vol. 26, no. 7, pp. 106–113, Apr. 1991. [Online]. Available: http://doi.acm.org/10.1145/109626.109637

[198] P. E. McKenney and J. Walpole, "What is RCU, fundamentally?" December 2007, available: http://lwn.net/Articles/262464/ [Viewed December 27, 2007].

[199] P. McKenney, "The design of preemptible read-copy-update," October 2007, available: http://lwn.net/Articles/253651/ [Viewed October 25, 2007].

[200] P. E. McKenney, D. Sarma, I. Molnar, and S. Bhattacharya, "Extending rcu for realtime and embedded workloads," in *Ottawa Linux Symposium, pages v2*, 2006, pp. 123–138.

[201] P. E. McKenney and D. Sarma, "Adapting rcu for real-time operating system usage," Oct. 23 2007, uS Patent 7,287,135.

[202] P. E. McKenney, "Sleepable RCU," October 2006, available: http://lwn.net/Articles/202847/ Revised: http://www.rdrop.com/users/paulmck/RCU/srcu.2007.01.14a.pdf [Viewed August 21, 2006].

[203] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole, "The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux," *IBM Systems Journal*, vol. 47, no. 2, pp. 221–236, May 2008.

[204] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole, "User-level implementations of read-copy update," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 2, pp. 375–382, 2012.

[205] M. Desnoyers, "[RFC git tree] userspace RCU (urcu) for Linux," February 2009, http://lttng.org/urcu.

[206] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole, "Performance of memory reclamation for lockless synchronization," *J. Parallel Distrib. Comput.*, vol. 67, no. 12, pp. 1270–1285, 2007.

[207] P. E. McKenney. (2016) Read-copy update (rcu) for c++. [Online]. Available: http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0279r0.html

[208] P. Goodman. (2018) C++ implementation of rcu based on reference counting and hazard pointers. [Online]. Available: https://github.com/pgoodman/rcu

[209] M. Khizhinsky. (2018) A c++ library of concurrent data structures. [Online]. Available: https://github.com/khizmax/libcds

[210] H. Sutter, "Atomic smart pointers, rev. 1," ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Tech. Rep. n4162, Oct. 2014.

[211] stackoverflow.com, "Understanding std::atomic::compare_exchange_weak() in c++11," 2017. [Online]. Available: https://goo.gl/jwjgGC

[212] R. K. Treiber, *Systems programming: Coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[213] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Understanding and effectively preventing the aba problem in descriptor-based lock-free designs," in *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on.* IEEE, 2010, pp. 185–192.

[214] A. Williams, "Why do we need atomic_shared_ptr?" August 2015, available: https://www.justsoftwaresolutions.co.uk/threading/why-do-we-need-atomic_shared_ptr.html.

[215] Anthony Williams, *C++ concurrency in action: practical multithreading.* Manning Publ., 2012.

[216] stackoverflow.com, "Why are std::atomic objects not copyable?" 2017. [Online]. Available: https://goo.gl/fvuY3f

[217] P. E. McKenney, T. Riegel, J. Preshing, H. Boehm, C. Nelson, O. Giroux, and L. Crowl, "Towards implementation and use of memory_order_consume," ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Tech. Rep. P0098R0, 2015.

[218] H.-J. Boehm, "Temporarily deprecate memory_order_consume," ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Tech. Rep. P0371R0, May 2016.

[219] M. McCarty, "Implementing a lock-free atomic_shared_ptr," 2016, cppNow 2016. [Online]. Available: https://goo.gl/qErf1h

[220] A. Williams, "Implementation of a lock-free atomic_shared_ptr class template as described in n4162," 2016. [Online]. Available: https://bitbucket.org/anthonyw/atomic_shared_ptr

[221] D. Hart, "A futex overview and update," *LWN. net*, 2009.

[222] Intel®, "Intel® threading building blocks documentation," 2018. [Online]. Available: https://software.intel.com/en-us/tbb-documentation

[223] M. Desnoyers, "Proving the correctness of nonblocking data structures," *Communications of the ACM*, vol. 56, no. 7, pp. 62–69, 2013.

[224] Intel. (2017) Pin user guide. [Online]. Available: https://software.intel.com/sites/landingpage/pintool/docs/97438/Pin/html/

[225] llvm.org. (2019) The llvm compiler infrastructure. Https://llvm.org. [Online]. Available: https://llvm.org/

[226] llvm.org. (2019) Clang documentation. Https://clang.llvm.org/docs/index.html. [Online]. Available: https://clang.llvm.org/docs/index.html

Appendices

Appendix A

Intel Pin: a Run-time FCI Seam

Intel Pin is a run-time FCI framework which can be exploited to create non-intrusive tests. (The various FCI techniques are described in detail in section 2.3.1.) Intel Pin is essentially a "just in time" (JIT) compiler [224, 76]. The input to this compiler is not bytecode, however, but a regular executable. Pin intercepts the execution of the first instruction of the executable and generates ("compiles") new code for the straight line code sequence starting at this instruction. It then transfers control to the generated sequence. The generated code sequence is almost identical to the original one, but Pin ensures that it regains control when a branch exits the sequence. After regaining control, Pin generates more code for the branch target and continues execution. Pin supports two modes of instrumentation: JIT mode and Probe mode. JIT mode uses a just-in-time compiler to recompile all program code and insert instrumentation, while Probe mode uses code trampolines for instrumentation.

In JIT mode, the only code ever executed is the generated code. The original code is only used for reference. When generating code, Pin gives the user an opportunity to inject their own code (instrumentation). Pin instruments all instructions that are actually executed. It does not matter what section they reside in. Although there are some exceptions for conditional branches, generally speaking, if an instruction is never executed then it will not be instrumented.

Probe mode is a method of using Pin to insert probes at the start of specified routines. A probe is a jump instruction that is placed at the start of the specified routine. The probe redirects the flow of control to the replacement function. Before the probe is inserted, the first few instructions of the specified routine are relocated. In probe mode, the application and the replacement routine are run natively; this improves performance. It is the tool writer's responsibility to ensure that no thread is currently executing the code where a probe is inserted. Tool writers are encouraged to insert probes when an image is loaded to avoid this problem.

Let us consider the legacy graphics program presented in Figure 2.5 and described in details in 2.3.1. The `Painter` class has a hard-wired dependency on the concrete `Turtle` class. Our goal is to write a non-intrusive test with the help of Intel Pin. Figure A.1 lists the non-intrusive test application which we will instrument with Pin to set up the function replacements. The code for this test application is very similar to the code we used in Figure 2.6 in case of the compile-time FCI seam. However, there is a very important difference: this time we cannot use the `SUBSTITUTE` macro, so the body of the `TurtleTest` fixture class is empty. Instead of the `SUBSTITUTE` macro, we will use Pin to set up the replacement functions for each member functions of the `Turtle` class. Similarly to the compile-time FCI based seam, the replacement functions behave like a proxy; they forward each function call on a given `Turtle` instance to a corresponding test double.

In Figure A.2 we present the Pin tool which we use to set up the function replacements. In `main()` we initialize the Pin runtime system, register the `ImageLoad()` function to be called when an image is loaded and we start the tool in probe mode. The `ImageLoad()` function is called when a shared library or the image of the test application is loaded into memory. We try to find the `Turtle::PenDown()` function by its mangled name (lines 7-8). If the loaded image contains the function then we replace the function – in probed mode – with the `proxy::PenDown(Turtle*)` function (lines 9-14). Note, the signature of the original and the substituted function differ, but that is not a problem. In order to be able to replace inline functions, the test application must be built with inlining disabled (e.g. with `-fno-inline-functions`).

If we executed the test binary as it is – i.e. without involving the Pin tool into the execution – then the test would fail.

```cpp
#include "Turtle.hpp"
#include <gmock/gmock.h>
#include <access_private.hpp>
#include <hook.hpp> // for SUBSTITUTE

class MockTurtle {
public:
  MOCK_METHOD0(PenUp, void());
  // PenDown, Forward, ...
};

MockTurtle& GetMockObject(Turtle*) {
    static MockTurtle m;
    return m;
}

namespace proxy {
    void PenUp(Turtle* self) {
        return GetMockObject(self).PenUp();
    }
    // Similarly to PenDown, Forward, ...
}

ACCESS_PRIVATE_FIELD(Painter, Turtle, turtle)

TEST_F(TurtleTest, TestDrawLine) {
    using ::testing::AtLeast;

    Painter painter;
    Turtle& turtle = access_private::turtle(painter);
    MockTurtle& mockTurtle = GetMockObject(&turtle);

    EXPECT_CALL(mockTurtle, PenDown())
        .Times(AtLeast(1));
    painter.DrawLine(0, 0, 10, 10);
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Figure A.1: Test application for the legacy program

```
#include "pin.H"
#include <iostream>

VOID ImageLoad(IMG img, VOID * v) {
  // Replace Turtle::PenDown() with
  // proxy::PenDown(Turtle*)
  RTN rtn =
      RTN_FindByName(img, "_ZN6Turtle7PenDownEv");
  if (RTN_Valid(rtn)) {
    RTN_ReplaceProbed(
        rtn,
        RTN_Funptr(RTN_FindByName(
            img, "_ZN5proxy7PenDownEP6Turtle")));
  }

  // Replace other member functions of Turtle
  // ...
}

int main(INT32 argc, CHAR * argv[]) {

  PIN_InitSymbols();
  if (PIN_Init(argc, argv))
    return -1;

  IMG_AddInstrumentFunction(ImageLoad, 0);

  PIN_StartProgramProbed();
  return 0;
}
```

Figure A.2: Pin tool to replace functions in the test application

We would receive the following failure:

```
turtle.cpp:57: Failure
Actual function call count doesn't match
EXPECT_CALL(mockTurtle, PenDown())...
  Expected: to be called at least once
    Actual: never called - unsatisfied and active
```

However, if we execute the binary with the Pin tool,

```
> pin -t TurtleReplaceFun.so -- turtle_test
```

then we no longer receive this failure and the test passes. Note that `TurtleReplaceFun.so` is the Pin tool and `turtle_test` is the test application.

There are the following disadvantages if we use Intel Pin as a test seam for non-intrusive testing:

- The most obvious drawback is that the test setup (i.e. where we do the function replacements) is entirely separated from the test code. In the context of the given-when-then pattern (see 2.2.2), the "given" part is isolated from the rest of the test, thus it violates the pattern.
- Some platforms are not supported at all. For instance, replacing functions with Pin is not possible on macOS.
- We need a completely new tool (Pin) to be introduced into the existing build chain. Developers and maintainers of the project must learn and understand Pin.
- We must use mangled names. This means we need some additional software tool to receive the mangled names for a specific platform.
- The setup or tear-down of different test case may require clearing the substitutions of the functions. There is no obvious way to clear all the function replacements, therefore the easiest way is to have a new Pin tool for each test case. To launch a new process for each test cases may decrease the run time of the test suite compared to the case where all test cases are in one process.
- To replace inline functions in the application binary, we have to rebuild the application with inlining disabled.

Appendix B

The LLVM/Clang Compiler Infrastructure

The measurements and prototypes presented in this dissertation are implemented with the help of the LLVM/Clang compiler infrastructure. In this appendix, we give an excerpt of the (online available) documentation of the infrastructure and its usage. We cannot describe the entire framework because of its extent, rather we focus on those methods which we used frequently. The extensive and thorough documentation is available online at [225, 82, 226, 83, 84].

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project. LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of subprojects, many of which are being used in production by a wide variety of commercial and open source projects as well as being widely used in academic research.

Some of the LLVM sub-projects are:

- The *LLVM Core* libraries provide a modern source- and target-independent optimizer, along with code generation support for many popular CPUs. These libraries are built around a well-specified code representation known as the LLVM intermediate representation ("LLVM IR"). The LLVM Core libraries are well documented, and it is particularly easy to invent a new language (or port an existing compiler) to use LLVM as an optimizer and code generator.

- *Clang* is an "LLVM native" C/C++/Objective-C compiler, which aims to

deliver amazingly fast compiles, extremely useful error and warning messages and to provide a platform for building great source level tools.

- The compiler-rt project provides highly tuned implementations of the low-level code generator support routines and other calls generated when a target doesn't have a short sequence of native instructions to implement a core IR operation. It also provides implementations of run-time libraries for dynamic testing tools such as AddressSanitizer, ThreadSanitizer, MemorySanitizer, and DataFlowSanitizer.

B.1 Introduction to the LLVM IR

This section is a very brief extract of the LLVM language reference focusing only on the types and instructions used in this dissertation. The full reference is available at [83].

LLVM is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly. It is the common code representation used throughout all phases of the LLVM compilation strategy.

LLVM identifiers come in two basic types: global and local. Global identifiers (functions, global variables) begin with the `'@'` character. Local identifiers (register names, types) begin with the `'%'` character. Additionally, there are three different formats for identifiers, for different purposes:

1. Named values are represented as a string of characters with their prefix. For example, `%foo`, `@DivisionByZero`, `%a.really.long.identifier`. The actual regular expression used is `'[%@][-a-zA-Z$._][-a-zA-Z$._0-9]*'`.

2. Unnamed values are represented as an unsigned numeric value with their prefix. For example, `%12`, `@2`, `%44`.

3. Constants.

Reserved words in LLVM are very similar to reserved words in other languages. There are keywords for different opcodes ('add', 'bitcast', 'ret', etc...), for primitive type names ('void', 'i32', etc...), and others. These reserved words cannot conflict with variable names, because none of them start with a prefix character (`'%'` or `'@'`).

Here is an example of LLVM code to multiply the integer variable `'%X'` by 8:

```
%result = mul i32 %X, 8    ; multiply by 8
```

Note that comments are delimited with a ';' and go until the end of the line.

B.1.1 Type System

The LLVM type system is one of the most important features of the intermediate representation. Being typed enables a number of optimizations to be performed on the intermediate representation directly.

The Integer Type

The integer type is a very simple type that simply specifies an arbitrary bit width for the integer type desired. The syntax `'iN'` represents a type where the number of bits the integer will occupy is specified by the N value. Examples:

- `i1` a single-bit integer.
- `i32` a 32-bit integer.

The Pointer Type

The pointer type is used to specify memory locations. Pointers are commonly used to reference objects in memory. Note that LLVM does not permit pointers to void (void*) nor does it permit pointers to labels (label*). Use i8* instead. Syntax:

```
<type> *
```

The Function Type

The function type can be thought of as a function signature. It consists of a return type and a list of formal parameter types. The return type of a function type is a void type or first class type. Syntax:

```
<returntype> (<parameter list>)
```

Examples:

- `i32 (i32)` function taking an i32, returning an i32.
- `i8 (i16, i32 *) *` Pointer to a function that takes an i16 and a pointer to i32, returning an i8.

B.1.2 Instructions

The 'bitcast .. to' Instruction

```
<result> = bitcast <ty> <value> to <ty2>             ; yields ty2
```

The 'bitcast' instruction converts value to type ty2 without changing any bits.

The 'icmp' Instruction

```
<result> = icmp <cond> <ty> <op1>, <op2>   ; yields i1
```

The 'icmp' instruction returns a boolean value based on comparison of its two integer or pointer operands. The 'icmp' instruction takes three operands. The first operand is the condition code indicating the kind of comparison to perform. It is not a value, just a keyword. The remaining two arguments must be integer or pointer typed. They must also be identical types.

The 'br' Instruction

```
br i1 <cond>, label <iftrue>, label <iffalse>
br label <dest>          ; Unconditional branch
```

The 'br' instruction is used to cause control flow to transfer to a different basic block in the current function. There are two forms of this instruction, corresponding to a conditional branch and an unconditional branch. Upon execution of a conditional 'br' instruction, the 'i1' argument is evaluated. If the value is true, control flows to the 'iftrue' label argument. If "cond" is false, control flows to the 'iffalse' label argument.

The 'phi' Instruction

```
<result> = phi <ty> [ <val0>, <label0>], ...
```

The 'phi' instruction is used to implement the phi node in the SSA graph representing the function. The type of the incoming values is specified with the first type field. After this, the 'phi' instruction takes a list of pairs as arguments, with one pair for each predecessor basic block of the current block. For the purposes of the SSA form, the use of each incoming value is deemed to occur on the edge from the corresponding predecessor block to the current block.

B.2 Introduction to the Clang AST

Clang's AST is different from ASTs produced by some other compilers in that it closely resembles both the written C++ code and the C++ standard. For example, parenthesis expressions and compile-time constants are available in an unreduced form in the AST. This makes Clang's AST a good fit for refactoring tools. Clang has a builtin AST-dump mode, which can be enabled with the flag `-ast-dump`. Let us look at a simple example AST:

```
$ cat test.cc
int f(int x) {
  int result = (x / 42);
  return result;
}

# Clang by default is a frontend for many tools; -Xclang is used to pass
# options directly to the C++ frontend.
$ clang -Xclang -ast-dump -fsyntax-only test.cc
TranslationUnitDecl 0x5aea0d0 <<invalid sloc>>
... cutting out internal declarations of clang ...
`-FunctionDecl 0x5aeab50 <test.cc:1:1, line:4:1> f 'int␣(int)'
  |-ParmVarDecl 0x5aeaa90 <line:1:7, col:11> x 'int'
  `-CompoundStmt 0x5aead88 <col:14, line:4:1>
    |-DeclStmt 0x5aead10 <line:2:3, col:24>
    | `-VarDecl 0x5aeac10 <col:3, col:23> result 'int'
    |   `-ParenExpr 0x5aeacf0 <col:16, col:23> 'int'
    |     `-BinaryOperator 0x5aeacc8 <col:17, col:21> 'int' '/'
    |       |-ImplicitCastExpr 0x5aeacb0 <col:17> 'int' <LValueToRValue>
    |       | `-DeclRefExpr 0x5aeac68 <col:17> 'int' lvalue ParmVar 0x5aeaa90 'x' 'int'
    |       `-IntegerLiteral 0x5aeac90 <col:21> 'int' 42
    `-ReturnStmt 0x5aead68 <line:3:3, col:10>
      `-ImplicitCastExpr 0x5aead50 <col:10> 'int' <LValueToRValue>
        `-DeclRefExpr 0x5aead28 <col:10> 'int' lvalue Var 0x5aeac10 'result' 'int'
```

The top-level declaration in a translation unit is always the translation unit declaration. In this example, our first user-written declaration is the function declaration of "f". The body of "f" is a compound statement, whose child nodes are a declaration statement that declares our result variable, and the return statement.

All information about the AST for a translation unit is bundled up in the class *ASTContext*. It allows traversal of the whole translation unit starting from `TranslationUnitDecl`.

Clang's AST nodes are modelled on a class hierarchy that does not have a common ancestor. Instead, there are multiple larger hierarchies for basic node types like declarations (Decl) and statements (Stmt). Note that expressions (Expr) are also statements in Clang's AST. Many important AST nodes derive from Type, Decl, DeclContext or Stmt, with some classes deriving from both Decl and DeclContext.

There are also a multitude of nodes in the AST that are not part of a larger hierarchy and are only reachable from specific other nodes, like CXXBaseSpecifier (which represents a base class of a C++ class).

Thus, to traverse the full AST, one starts from the TranslationUnitDecl and then recursively traverses everything that can be reached from that node - this information has to be encoded for each specific node type. This algorithm is encoded in the *RecursiveASTVisitor*. We used the RecursiveASTVisitor API to measure the usage statistics of friends in Chapter 3.

B.3 RecursiveASTVisitor

In this section, we describe how to create a FrontendAction that uses a RecursiveASTVisitor to find CXXRecordDecl AST nodes (which represent C++ classes)

with a specified name. When writing a clang based tool like a Clang Plugin or a standalone tool based on LibTooling, the common entry point is the *FrontendAction*. FrontendAction is an interface that allows execution of user-specific actions as part of the compilation. To run tools over the AST clang provides the convenience interface ASTFrontendAction, which takes care of executing the action. The only part left is to implement the CreateASTConsumer method that returns an ASTConsumer per translation unit.

```
class FindNamedClassAction : public clang::ASTFrontendAction {
public:
  virtual std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
    clang::CompilerInstance &Compiler, llvm::StringRef InFile) {
    return std::unique_ptr<clang::ASTConsumer>(
        new FindNamedClassConsumer);
  }
};
```

ASTConsumer is an interface used to write generic actions on an AST, regardless of how the AST was produced. ASTConsumer provides many different entry points, but for our use case, the only one needed is HandleTranslationUnit, which is called with the ASTContext for the translation unit.

```
class FindNamedClassConsumer : public clang::ASTConsumer {
public:
  virtual void HandleTranslationUnit(clang::ASTContext &Context) {
    // Traversing the translation unit decl via a RecursiveASTVisitor
    // will visit all nodes in the AST.
    Visitor.TraverseDecl(Context.getTranslationUnitDecl());
  }
private:
  // A RecursiveASTVisitor implementation.
  FindNamedClassVisitor Visitor;
};
```

Now that everything is hooked up, the next step is to implement a RecursiveASTVisitor to extract the relevant information from the AST. The RecursiveASTVisitor provides hooks of the form bool VisitNodeType(NodeType *) for most AST nodes; the exception are TypeLoc nodes, which are passed by value. We only need to implement the methods for the relevant node types. Let us start by writing a RecursiveASTVisitor that visits all CXXRecordDecl's.

```
class FindNamedClassVisitor
  : public RecursiveASTVisitor<FindNamedClassVisitor> {
public:
  bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
    // For debugging, dumping the AST nodes will show which nodes are already
    // being visited.
    Declaration->dump();

    // The return value indicates whether we want the visitation to proceed.
    // Return false to stop the traversal of the AST.
    return true;
  }
};
```

In the methods of our RecursiveASTVisitor we can now use the full power of the Clang AST to drill through to the parts that are interesting for us. For example, to

find all class declaration with a certain name, we can check for a specific qualified name:

```
bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
  if (Declaration->getQualifiedNameAsString() == "n::m::C")
    Declaration->dump();
  return true;
}
```

Some of the information about the AST, like source locations and global identifier information, are not stored in the AST nodes themselves, but in the ASTContext and its associated source manager. To retrieve them we need to hand the ASTContext into our RecursiveASTVisitor implementation. The ASTContext is available from the CompilerInstance during the call to CreateASTConsumer. We can thus extract it there and hand it into our freshly created FindNamedClassConsumer:

```
virtual std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
  clang::CompilerInstance &Compiler, llvm::StringRef InFile) {
  return std::unique_ptr<clang::ASTConsumer>(
      new FindNamedClassConsumer(&Compiler.getASTContext()));
}
```

Now that the ASTContext is available in the RecursiveASTVisitor, we can do more interesting things with AST nodes, like looking up their source locations:

```
bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
  if (Declaration->getQualifiedNameAsString() == "n::m::C") {
    // getFullLoc uses the ASTContext's SourceManager to resolve the source
    // location and break it up into its line and column parts.
    FullSourceLoc FullLocation = Context->getFullLoc(Declaration->getBeginLoc());
    if (FullLocation.isValid())
      llvm::outs() << "Found␣declaration␣at␣"
                   << FullLocation.getSpellingLineNumber() << ":"
                   << FullLocation.getSpellingColumnNumber() << "\n";
  }
  return true;
}
```

Now we can combine all of the above into a small example program which is represented in Figure B.1. We store this into a file called FindClassDecls.cpp and create the following CMakeLists.txt to link it:

```
add_clang_executable(find-class-decls FindClassDecls.cpp)

target_link_libraries(find-class-decls clangTooling)
```

When running this tool over a small code snippet it will output all declarations of a class n::m::C it found:

```
$ ./bin/find-class-decls "namespace n { namespace m { class C {}; } }"
Found declaration at 1:29
```

```
#include "clang/AST/ASTConsumer.h"
#include "clang/AST/RecursiveASTVisitor.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendAction.h"
#include "clang/Tooling/Tooling.h"

using namespace clang;

class FindNamedClassVisitor
  : public RecursiveASTVisitor<FindNamedClassVisitor> {
public:
  explicit FindNamedClassVisitor(ASTContext *Context)
    : Context(Context) {}

  bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
    if (Declaration->getQualifiedNameAsString() == "n::m::C") {
      FullSourceLoc FullLocation = Context->getFullLoc(Declaration->getBeginLoc());
      if (FullLocation.isValid())
        llvm::outs() << "Found␣declaration␣at␣"
                     << FullLocation.getSpellingLineNumber() << ":"
                     << FullLocation.getSpellingColumnNumber() << "\n";
    }
    return true;
  }

private:
  ASTContext *Context;
};

class FindNamedClassConsumer : public clang::ASTConsumer {
public:
  explicit FindNamedClassConsumer(ASTContext *Context)
    : Visitor(Context) {}

  virtual void HandleTranslationUnit(clang::ASTContext &Context) {
    Visitor.TraverseDecl(Context.getTranslationUnitDecl());
  }
private:
  FindNamedClassVisitor Visitor;
};

class FindNamedClassAction : public clang::ASTFrontendAction {
public:
  virtual std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
    clang::CompilerInstance &Compiler, llvm::StringRef InFile) {
    return std::unique_ptr<clang::ASTConsumer>(
        new FindNamedClassConsumer(&Compiler.getASTContext()));
  }
};

int main(int argc, char **argv) {
  if (argc > 1) {
    clang::tooling::runToolOnCode(new FindNamedClassAction, argv[1]);
  }
}
```

Figure B.1: Clang tool to traverse the AST

B.4 Adding a New Attribute to Clang

Several of prototypes of this dissertation are implemented by adding a new attribute to Clang, thus, in this section, we overview how we can extend the compiler in this way. Attributes are a form of metadata that can be attached to a program construct, allowing the programmer to pass semantic information along to the compiler for various uses. For example, attributes may be used to alter the code generation for a program construct or to provide extra semantic information for static analysis. This document explains how to add a custom attribute to Clang.

Attributes in Clang are handled in three stages: parsing into a parsed attribute representation, conversion from a parsed attribute into a semantic attribute, and then the semantic handling of the attribute.

Parsing of the attribute is determined by the various syntactic forms attributes can take, such as GNU, C++11, and Microsoft style attributes, as well as other information provided by the table definition of the attribute. Ultimately, the parsed representation of an attribute object is a ParsedAttr object. These parsed attributes chain together as a list of parsed attributes attached to a declarator or declaration specifier. The parsing of attributes is handled automatically by Clang, except for attributes spelt as keywords. When implementing a keyword attribute, the parsing of the keyword and creation of the ParsedAttr object must be done manually.

Eventually, `Sema::ProcessDeclAttributeList()` is called with a Decl and a ParsedAttr, at which point the parsed attribute can be transformed into a semantic attribute. The process by which a parsed attribute is converted into a semantic attribute depends on the attribute definition and semantic requirements of the attribute. The end result, however, is that the semantic attribute object is attached to the Decl object, and can be obtained by a call to `Decl::getAttr<T>()`.

The structure of the semantic attribute is also governed by the attribute definition given in Attr.td. This definition is used to automatically generate functionality used for the implementation of the attribute, such as a class derived from clang::Attr, information for the parser to use, automated semantic checking for some attributes, etc.

The first step to adding a new attribute to Clang is to add its definition to `include/clang/Basic/Attr.td`. This definition must derive from the Attr type or one of its derivatives. Most attributes will derive from the InheritableAttr type, which specifies that the attribute can be inherited by later redeclarations of the Decl it is associated with. InheritableParamAttr is similar to InheritableAttr, except that the attribute is written on a parameter instead of a declaration. If the attribute is intended to apply to a type instead of a declaration, such an attribute should derive from TypeAttr, and will generally not be given an AST representation. An attribute that inherits from IgnoredAttr is parsed but will

generate an ignored attribute diagnostic when used, which may be useful when an attribute is supported by another vendor but not supported by clang.

The definition will specify several key pieces of information, such as the semantic name of the attribute, the spellings the attribute supports, the arguments the attribute expects, and more. Most members of the Attr tablegen type do not require definitions in the derived definition as the default suffice. However, every attribute must specify at least a spelling list, a subject list, and a documentation list.

Spellings All attributes are required to specify a *spelling* list that denotes the ways in which the attribute can be spelt. For instance, a single semantic attribute may have a keyword spelling, as well as a C++11 spelling and a GNU spelling. An empty spelling list is also permissible and may be useful for attributes which are created implicitly. Some of these spellings are:

- `GCC` Spelt with a GNU-style `__attribute__((attr))` syntax and placement.
- `CXX11` Spelt with a C++-style `[[attr]]` syntax.

Subjects Attributes appertain to one or more Decl *subjects*. If the attribute attempts to attach to a subject that is not in the subject list, a diagnostic is issued automatically. Whether the diagnostic is a warning or an error depends on how the attribute's SubjectList is defined, but the default behaviour is to warn. The diagnostics displayed to the user are automatically determined based on the subjects in the list. By default, all subjects in the SubjectList must either be a Decl node defined in DeclNodes.td, or a statement node defined in StmtNodes.td.

Arguments Attributes may optionally specify a list of *arguments* that can be passed to the attribute. Attribute arguments specify both the parsed form and the semantic form of the attribute. For example, if `Args` is

```
[StringArgument<"Arg1">, IntArgument<"Arg2">]
```

then

```
__attribute__((myattribute("Hello", 3)))
```

will be a valid use; it requires two arguments while parsing, and the Attr subclass' constructor for the semantic attribute will require a string and integer argument. All arguments have a name and a flag that specifies whether the argument is optional. The associated C++ type of the argument is determined by the argument definition type.

Semantic handling All semantic processing of declaration attributes happens in `SemaDeclAttr.cpp`, and generally starts in the `ProcessDeclAttribute()` function. If the attribute is a "simple" attribute – meaning that it requires no custom semantic processing aside from what is automatically provided, we have to add a call to `handleSimpleAttribute<OurAttr>(S, D, Attr)` to the switch statement. Otherwise, we write a new `handleOurAttr()` function, and we add that to the switch statement.

Unless otherwise specified by the attribute definition, common semantic checking of the parsed attribute is handled automatically. This includes diagnosing parsed attributes that do not appertain to the given Decl, ensuring the correct minimum number of arguments are passed, etc. Most attributes are implemented to have some effect on the compiler. For instance, to modify the way code is generated, or to add extra semantic checks for an analysis pass, etc. Having added the attribute definition and conversion to the semantic representation for the attribute, what remains is to implement the custom logic requiring use of the attribute. The Decl object can be queried for the presence or absence of an attribute using `hasAttr<T>()`. To obtain a pointer to the semantic representation of the attribute, `getAttr<T>()` may be used.

Appendix C

Dissertation Summaries

C.1 Dissertation Summary

This dissertation presents novel research results in three fundamental areas of software development: testing, encapsulation and abstraction. The first two theses are centered around non-intrusive testing, a testing technique which does not require any structural modification in the production code. We discuss the existing non-intrusive testing methods and we enlist their advantages and disadvantages. We introduce a new method which is based on function call interception and has numerous clear benefits compared to preexisting efforts. With this method we can replace functions with test doubles even if they are inline functions. We describe two new additional and experimental approaches which make it possible to substitute types with test double types: one which exploits syntax tree transformations and another which is based on compile-time reflection. We demonstrate that often it is needed to access private members in order to have non-intrusive tests. We introduce new techniques to access private members of a class for the purpose of non-intrusive or white-box testing: a library based on explicit template instantiation and out-of-class friends.

Regarding encapsulation, we demonstrate that certain language constructs like the `friend` in C++ may provide exaggerated access to the internals of a class. This excessive access may be the source of errors in the software. We suggest a new language construct which makes it possible to restrict access of a friend only to a certain well specified set of members, this way it strengthens encapsulation and information hiding.

Besides encapsulation, abstraction plays an essential role in large scale software system development, especially when multiple threads of execution are involved. We demonstrate a new high-level abstraction for the read-copy-update concurrency pattern, which provides reasonable performance meanwhile it gives a generic and safe to use C++ API.

We provide a proof of concept implementation for all new techniques with one exception (the reflection based non-intrusive technique).

C.2. Disszertáció Összefoglaló

A disszertáció új kutatási eredményeket mutat be három alapvető szoftver fejlesztési területen: tesztelés, egységbezárás és absztrakció. Az első három tézis az ún. nem-tolakodó teszteléssel foglalkozik, amely egy olyan tesztelési technika mely során nem szükséges semmilyen strukturális módosítást végrehajtanunk a termék forráskódján. Megvitatjuk a már létező nem-tolakodó tesztelési módszereket és felsoroljuk ezek előnyeit és hátrányait. Bevezetünk egy új, nem-tolakodó tesztelési módszert amely függvény hívás közbeavatkozáson alapszik és számos egyértelmű előnnyel rendelkezik a korábbi megoldásokhoz képest. Ezzel az új technikával képesek vagyunk függvényeket teszt dublőrökkel helyettesíteni még akkor is ha azok inline függvények. Továbbá bemutatunk két új kísérleti eljárást amelyek lehetővé teszik, hogy akár típusokat is helyettesítsünk teszt dublőrökkel: az egyik metódus szintaxis fa transzformációkon alapszik, a másik pedig fordítási idejű reflectionön. Demonstráljuk, hogy gyakran előfordul, hogy szükséges privát tagokhoz hozzáférni a nem-tolakodó tesztek esetében. Bemutatunk két új módszert a privát tagok eléréséhez (és ily módon támogatjuk a nem-tolakodó és fehér doboz tesztek létrehozását): egy program könyvtárat amely explicit sablon példányosításon alapszik, illetve az osztályon kívüli barát (friend) nyelvi elemet.

Az egységbezárással kapcsolatosan szemléltetjük, hogy bizonyos nyelvi konstrukciók mint a C++ barát (friend) túlzottan erős hozzáférést nyújthat egy osztály belső elemeihez. Ez a túlzott hozzáférés hibák forrása lehet az adott szoftverben. Javaslatot teszünk egy új nyelvi elem létrehozására amely lehetővé teszi, hogy megszorítsuk ezt a hozzáférést csupán néhány jól specifikált taghoz, ily módon erősítendő az egységbezárást és adatrejtést.

Az egységbezárás mellett az absztrakció a másik alapvető szereplő ha nagyméretű szoftverek fejlesztéséről van szó. Különösen, ha többszálú programokról beszélünk. Bemutatunk egy új magas szintű C++ absztrakciót mely a read-copy-update konkurrens programozási mintán alapszik és elfogadható teljesítményt nyújt amellett, hogy kellően generikus és biztonságos használni.

Az itt bemutatott új módszerek mindegyikéhez tartozik prototípus implementáció (ez alól kivételt képez a reflection alapú nem-tolakodó tesztelés ötlete).

Acronyms

ABI Application Binary Interface.

ACPI Advanced Configuration and Power Interface.

AFC Actual Friend Classes.

AFCR Actual Friend Class Relationships.

AFM Actual Friend Methods.

API Application Programming Interface.

AST Abstract Syntax Tree.

CAS Compare and Swap.

CBOF(Back) Coupling Complexity in the Backward direction for Friends.

CCFF(1) Complexity in the Forward Direction for Friends.

CPU Central Processing Unit.

CRTP Curiously Recurring Template Pattern.

DEC Digital Equipment Corporation.

DI Dependency Injection.

FCI Function Call Interception.

GCC GNU Compiler Collection.

GNU GNU's Not Unix!.

IR Intermediate Representation.

ISO International Organization for Standardization.

ITK Insight Segmentation and Registration Toolkit.

KiB Kibibyte, 1 kibibyte is 1024 bytes.

LIFT A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks.

LLVM Low Level Virtual Machine.

MAF Members Accessed by Friends.

MOC Meta Object Compiler.

MOP Meta Object Protocol.

MSVC Microsoft® Visual C++.

ODR One Definition Rule.

OOP Object-oriented Programming.

ORM Object Relational Mapping.

OS Operating System.

POD Plain Old Data.

POSIX Portable Operating System Interface.

QSBR Quiescent-State-Based Reclamation RCU.

RAII Resource Acquisition is Initialization.

RCU Read-Copy-Update.

RFFC(1) Response set For Friend Class.

RTTI Run-time Type Information.

SSA Static Single Assignment.

STL Standard Template Library.

SUT System Under Test.

TBB Threading Building Blocks (Intel®).

TDD Test Driven Development.

TU Translation Unit.

URCU Userspace Read-Copy-Update.

Glossary

abstract syntax tree A data structure which represents the hierarchical syntactic structure of the source program. During certain compilation stages the abstract syntax tree is being transformed.

abstraction Abstractions describe system components, the nature of interactions among the components and the patterns that guide the composition of components into systems. It provides a representation of features without including background or low-level details.

befriending class A struct/class which declares one or more friend function, friend function template, friend class or friend class template.

black-box testing A testing strategy which views the program as a black box. Its goal is to be completely unconcerned about the internal behaviour and structure of the program.

dependency injection A technique whereby one object supplies the dependencies of another object. A dependency is an object that can be used as a service. An injection is the passing of a dependency to a dependent object (a client) that would use it. Dependency injection is one realization – amongst many others – of dependency replacement.

dependency replacement An abstract overall concept which involves all various techniques of substituting the dependencies with test doubles.

encapsulation A fundamental concept in object-oriented programming which is used to hide the values or state of a structured data object inside a class. It implies distinction between the specification (interface) and the implementation of a class, thus minimizing the dependencies among separately-written modules.

friend A C++ language element which may enable access to specified functions and classes to the private members of another class.

function call interception A technique of intercepting function calls at program runtime. Without directly modifying the original code, it enables to undertake certain operations before and/or after the called function or even to replace the intercepted call.

instrumentation Instrumentation means the ability of an application to monitor or measure the level of a product's performance, to diagnose errors and to write trace information. It is implemented by additional instructions, which are injected either to the source code or to the binary executable.

intercession A kind of reflection during which the program is capable of modifying its structure or state.

intermediate representation The data structure or code used internally by a compiler or virtual machine to represent source code. An IR is designed to be conducive for further processing, such as optimization and translation.

introspection A kind of reflection during which the program is observing its own state and structure.

intrusive testing A testing approach which requires source code modification.

mock classes Mock classes are used in unit tests to substitute real dependencies of a unit. Mock classes are special kind of proxy classes.

non-intrusive testing A testing method which does not require any modification in the production code.

proxy classes Proxy classes are those classes which have the exact same interface as the original class, but the implementation of each member function could be different.

read-copy-update A concurrent design pattern which allows extremely low runtime overhead for readers.

reflection Reflection is the ability of a program to inspect or modify its own structure.

simple aggregate class A C++ struct with publicly available fields and without methods.

simple aggregate proxy class A simple aggregate class is a proxy class, if all of its fields have a proxy class type. The built in types like int or float are considered as proxy types.

static single assignment In compiler design, static single assignment form (often abbreviated as SSA form or simply SSA) is a property of an intermediate representation (IR), which requires that each variable is assigned exactly once, and every variable is defined before it is used.

system under test Refers to a system that is being tested for correct operation.

test seam A point in the software development ecosystem via which we can alter the behaviour of the system under test (SUT) without changing the production code of the SUT.

test-driven development A software development practice which requires writing automated tests prior to developing any functional code. The development consists of very short iterations of writing a new test and then providing implementation for it.

translation unit In C/C++ programming language terminology, a translation unit is the ultimate input to the compiler from which an object file is generated.

vpointer A data member silently inserted by the compiler into the class. During the construction of each object, it is initialized to point to the virtual table of the dynamic type.

vtable The virtual table is a lookup table of functions used to resolve function calls in a dynamic/late binding manner.

white-box testing A testing strategy which permits us to examine the internal structure of the program. This strategy derives test data from an examination of the program's logic.

[1]**ADATLAP**
a doktori értekezés nyilvánosságra hozatalához

I. A doktori értekezés adatai

A szerző neve: Márton Gábor

MTMT-azonosító: 10046083

A doktori értekezés címe és alcíme:

Tools and Language Elements for Testing, Encapsulation and Controlling Abstraction in Large Scale C++ Projects

DOI-azonosító[2]: 10.15476/ELTE.2019.084

A doktori iskola neve: Informatika Doktori Iskola

A doktori iskolán belüli doktori program neve: Az informatika alapjai és módszertana

A témavezető neve és tudományos fokozata: Porkoláb Zoltán, PhD

A témavezető munkahelye: Eötvös Loránd Tudományegyetem Informatikai Kar, Programozási Nyelvek és Fordítóprogramok Tanszék

II. Nyilatkozatok

1. A doktori értekezés szerzőjeként[3]

a) hozzájárulok, hogy a doktori fokozat megszerzését követően a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az ELTE Digitális Intézményi Tudástárban. Felhatalmazom az Informatika Doktori Iskola hivatalának ügyintézőjét, Kulcsár Adinát, hogy az értekezést és a téziseket feltöltse az ELTE Digitális Intézményi Tudástárba, és ennek során kitöltse a feltöltéshez szükséges nyilatkozatokat.

b) kérem, hogy a mellékelt kérelemben részletezett szabadalmi, illetőleg oltalmi bejelentés közzétételéig a doktori értekezést ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;[4]

c) kérem, hogy a nemzetbiztonsági okból minősített adatot tartalmazó doktori értekezést a minősítés (*dátum*)-ig tartó időtartama alatt ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;[5]

d) kérem, hogy a mű kiadására vonatkozó mellékelt kiadó szerződésre tekintettel a doktori értekezést a könyv megjelenéséig ne bocsássák nyilvánosságra az Egyetemi Könyvtárban, és az ELTE Digitális Intézményi Tudástárban csak a könyv bibliográfiai adatait tegyék közzé. Ha a könyv a fokozatszerzést követőn egy évig nem jelenik meg, hozzájárulok, hogy a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban.[6]

2. A doktori értekezés szerzőjeként kijelentem, hogy

a) az ELTE Digitális Intézményi Tudástárba feltöltendő doktori értekezés és a tézisek saját eredeti, önálló szellemi munkám és legjobb tudomásom szerint nem sértem vele senki szerzői jogait;

b) a doktori értekezés és a tézisek nyomtatott változatai és az elektronikus adathordozón benyújtott tartalmak (szöveg és ábrák) mindenben megegyeznek.

3. A doktori értekezés szerzőjeként hozzájárulok a doktori értekezés és a tézisek szövegének plágiumkereső adatbázisba helyezéséhez és plágiumellenőrző vizsgálatok lefuttatásához.

Kelt: Budapest, 2019.04.24

Márton Gábor

a doktori értekezés szerzőjének aláírása

---

[1] Beiktatta az Egyetemi Doktori Szabályzat módosításáról szóló CXXXIX/2014. (VI. 30.) Szen. sz. határozat. Hatályos: 2014. VII.1. napjától.

[2] A kari hivatal ügyintézője tölti ki.

[3] A megfelelő szöveg aláhúzandó.

[4] A doktori értekezés benyújtásával egyidejűleg be kell adni a tudományági doktori tanácshoz a szabadalmi, illetőleg oltalmi bejelentést tanúsító okiratot és a nyilvánosságra hozatal elhalasztása iránti kérelmet.

[5] A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a minősített adatra vonatkozó közokiratot.

[6] A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a mű kiadásáról szóló kiadói szerződést.